NO.37

编程狂人

Programming Madman

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/53e8c791d91b14209d00631b

欢迎下载推酷客户端体验更多阅读乐趣

目录

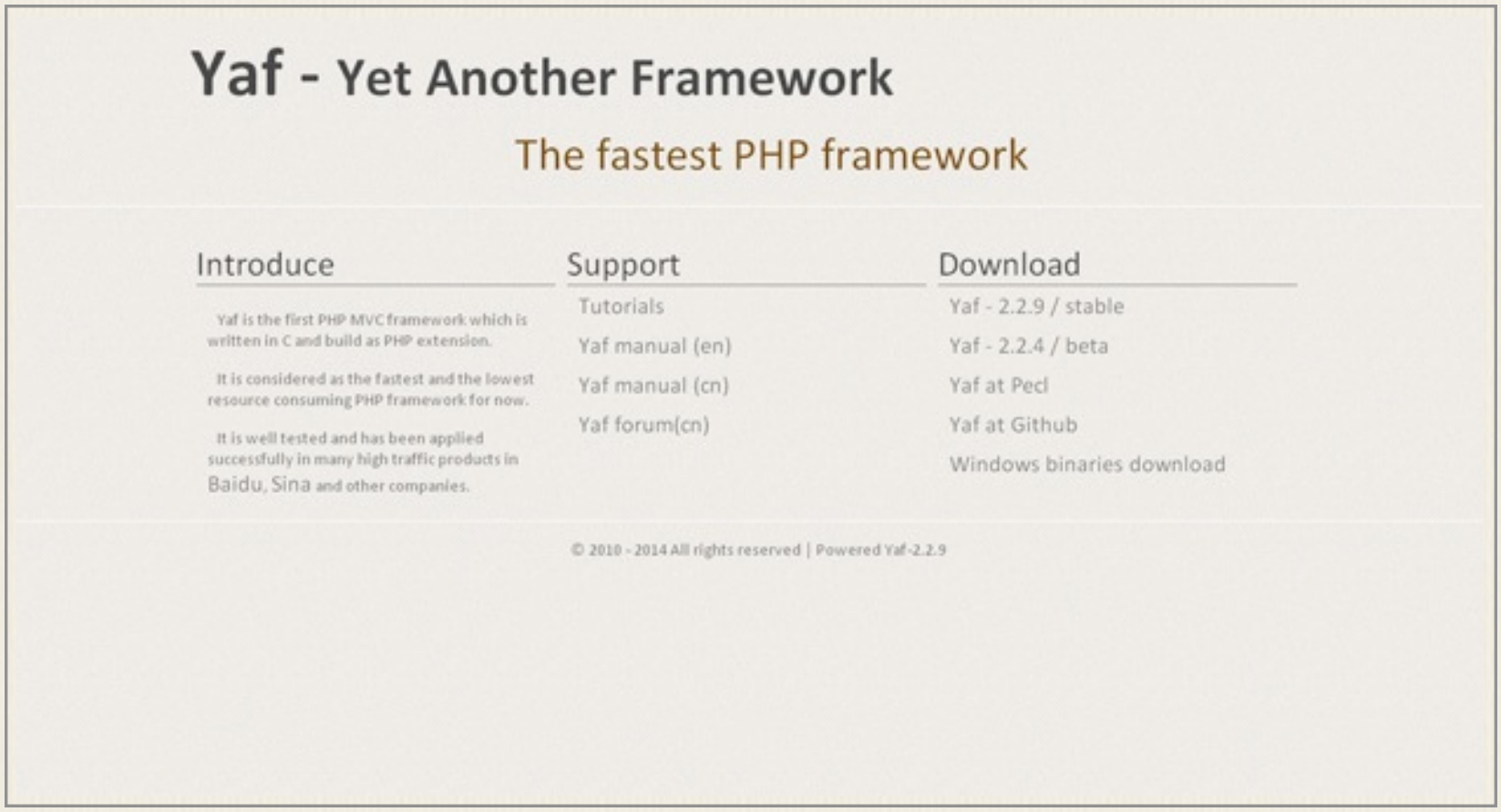
2014年15款新评定的最佳PHP框架

译者：oschina

通常，框架都会被认为是帮助开发者快速设计和开发动态网站的软件应用。每个月都有极大数量的新发布的 PHP 框架，使网站开发更简单更高效。

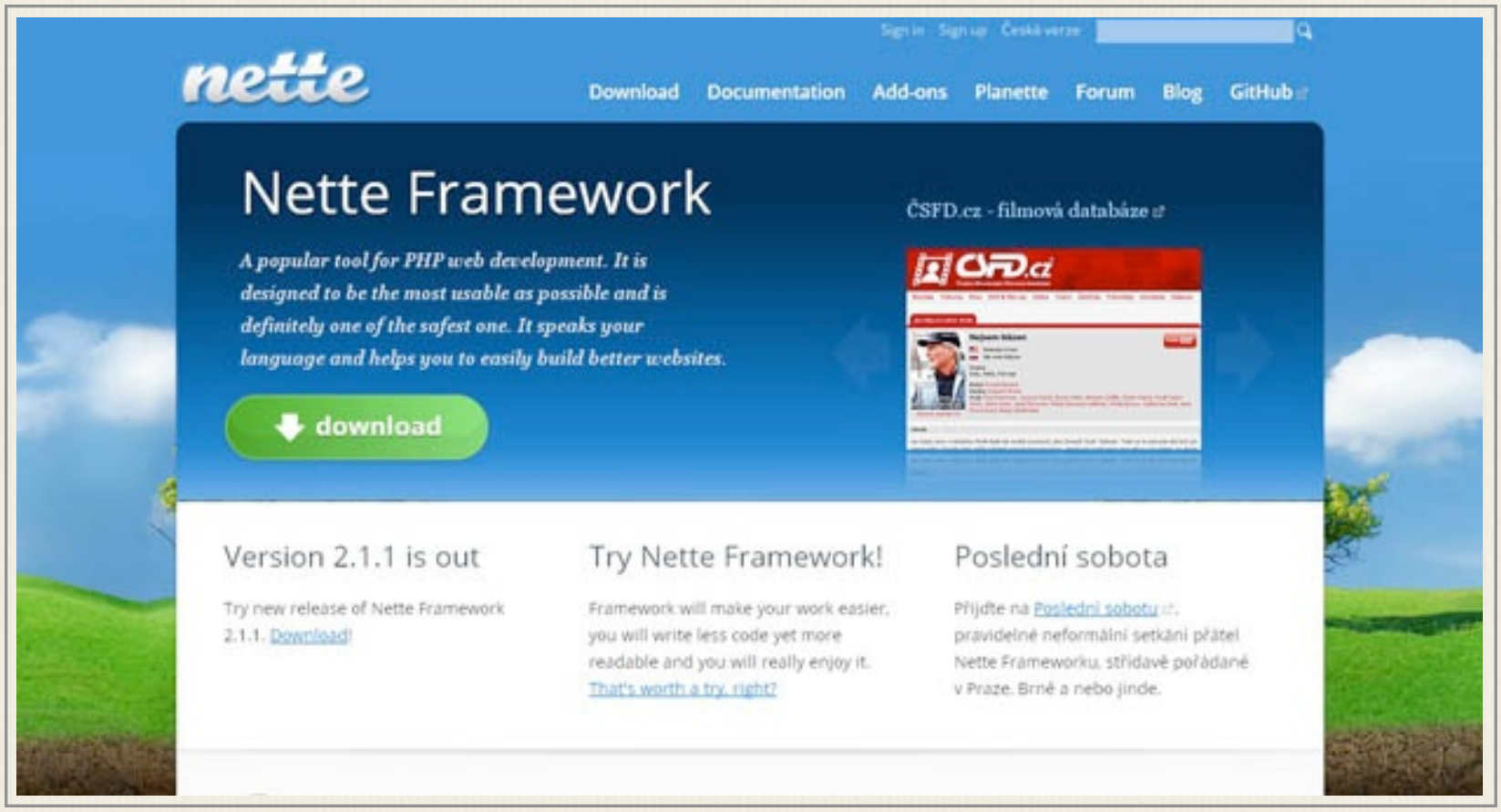
如果你是位 PHP 开发者，正在寻找当前最好的一些 PHP 框架来帮助开发你的项目，那么这里正是你要找的地方。在这篇文章我们会介绍 15 款最好的 PHP 框架，这些框架都是最新评定的，可以大大的简化你的开发任务。这些 PHP 框架可以帮助开发者快速设计和开发各种跨浏览器的动态网站和 web 应用，最后，希望你能在这些列表中找到你想要的 PHP 框架，Enjoy !!

1. Yaf : Yet Another Framework



Yaf 是第一个 PHP MVC 框架，用 C 语言编写，作为 PHP 的扩展来创建的。它被认为是最快和最低资源消耗的 PHP 框架，经过了良好的测试，并且现在已经很成功的应用在很多 web 项目上。

2. Nette Framework

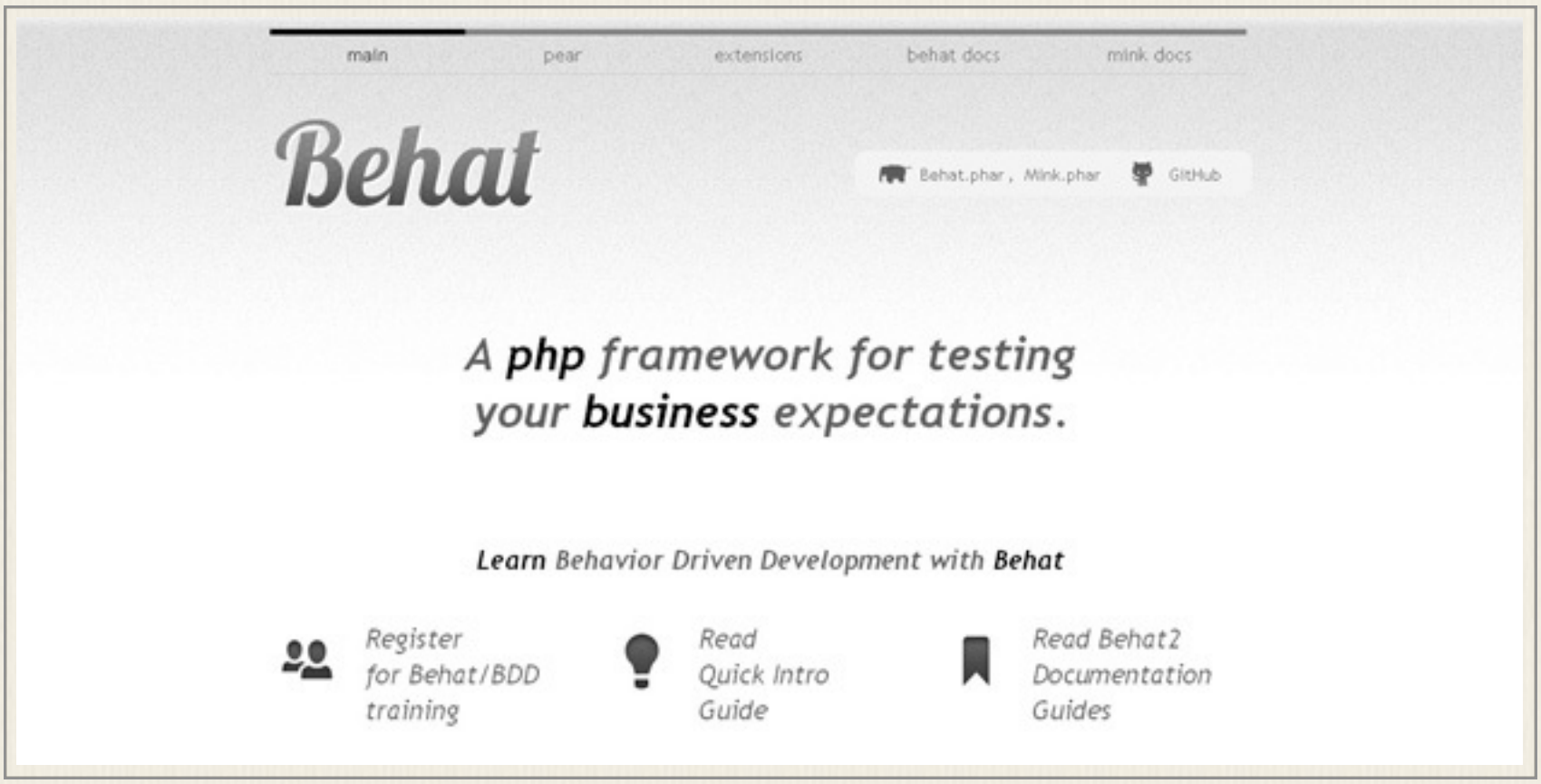


Nette Framework 是个现代化风格的 PHP 框架，对安全进行了革命性的改进，使用面向对象的设计理念，非一般的性能表现和超级简单的学习曲线。除了这些之外，它还有个非常活跃的社区，给予用户足够的灵活性。

3. Guzzle

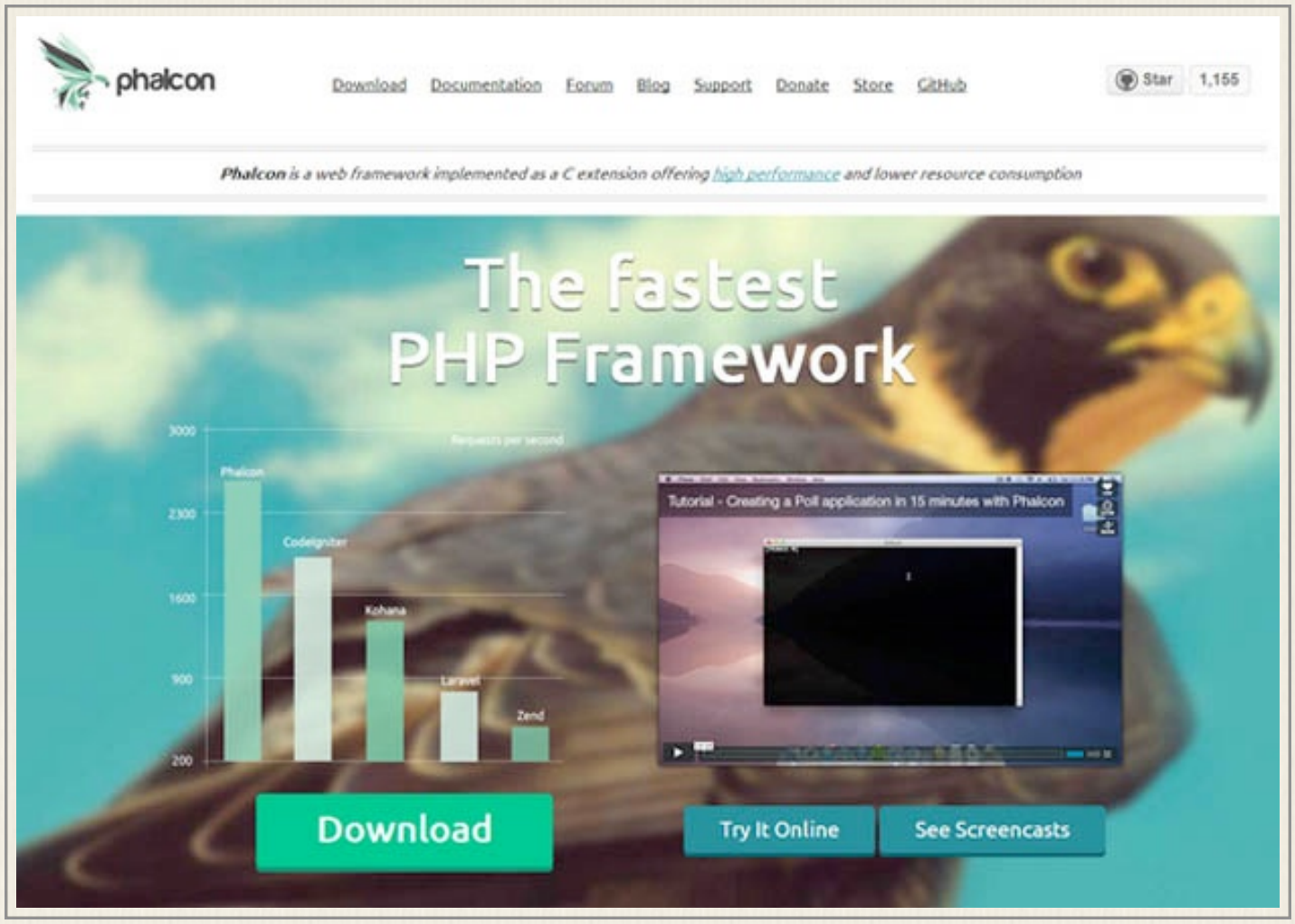
Guzzle 是个 PHP 框架，又是个 PHP HTTP 客户端，用来创建 RESTful web 服务客户端。它的主要特性是通过服务描述快速创建客户端；尽可能高效的批量发送大量的请求；持久性连接和并行请求；其他更多的功能。

4. Behat



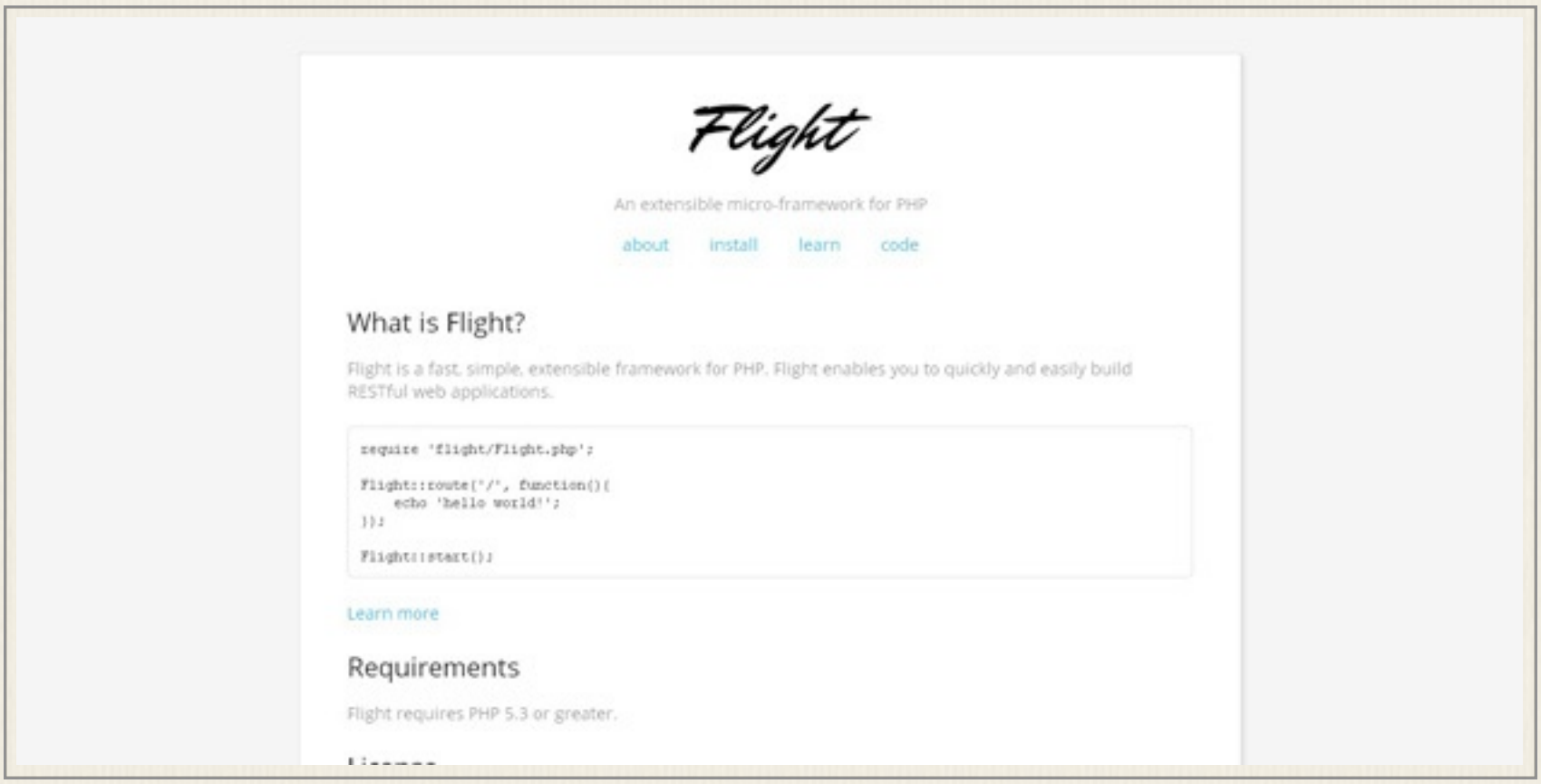
Behat 是个行为驱动的开发（BDD）框架，允许用户编写便于人们阅读的故事驱动代码，描述该应用应该怎样工作。任何人都能快速简单的掌握它的使用方法。

5. Phalcon



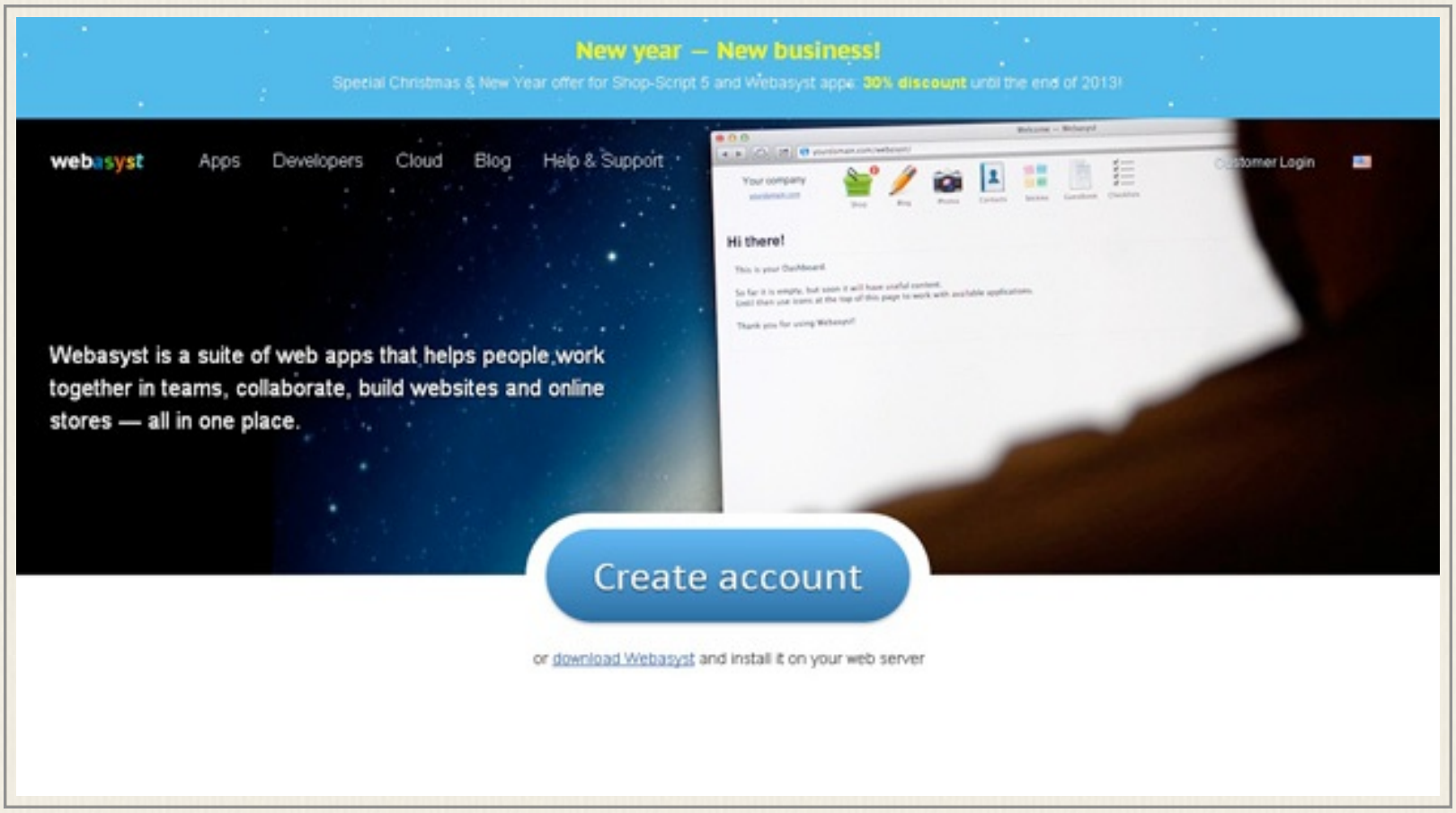
Phalcon 实现了 C 的扩展，是个高性能，低能耗 PHP框架。它包括一个模版引擎，加密，分页，assets 管理和其他更多的工具。

6. Flight



Flight 是个快速，简单，可扩展的 PHP 框架，允许用户快速简单的创建 RESTful web 应用。

7. Webasyst



Webasyst 是个开源的 PHP 框架，用来开发时尚的多用户 web 应用和高级的网站。相对比其他框架，比如 Zend 和 Symfony ，它更注重于构建商业和给团队使用的 web 应用，更快更高效。

8. Medoo

Medoo 是个轻量级的 PHP 数据库框架，帮助用户快速开发 web 应用。它支持多种 sql 数据库： MYSQL, MSSQL, SQLite, MariaDB 等等。它遵从 MIT 协议，允许用户在任何地方使用。

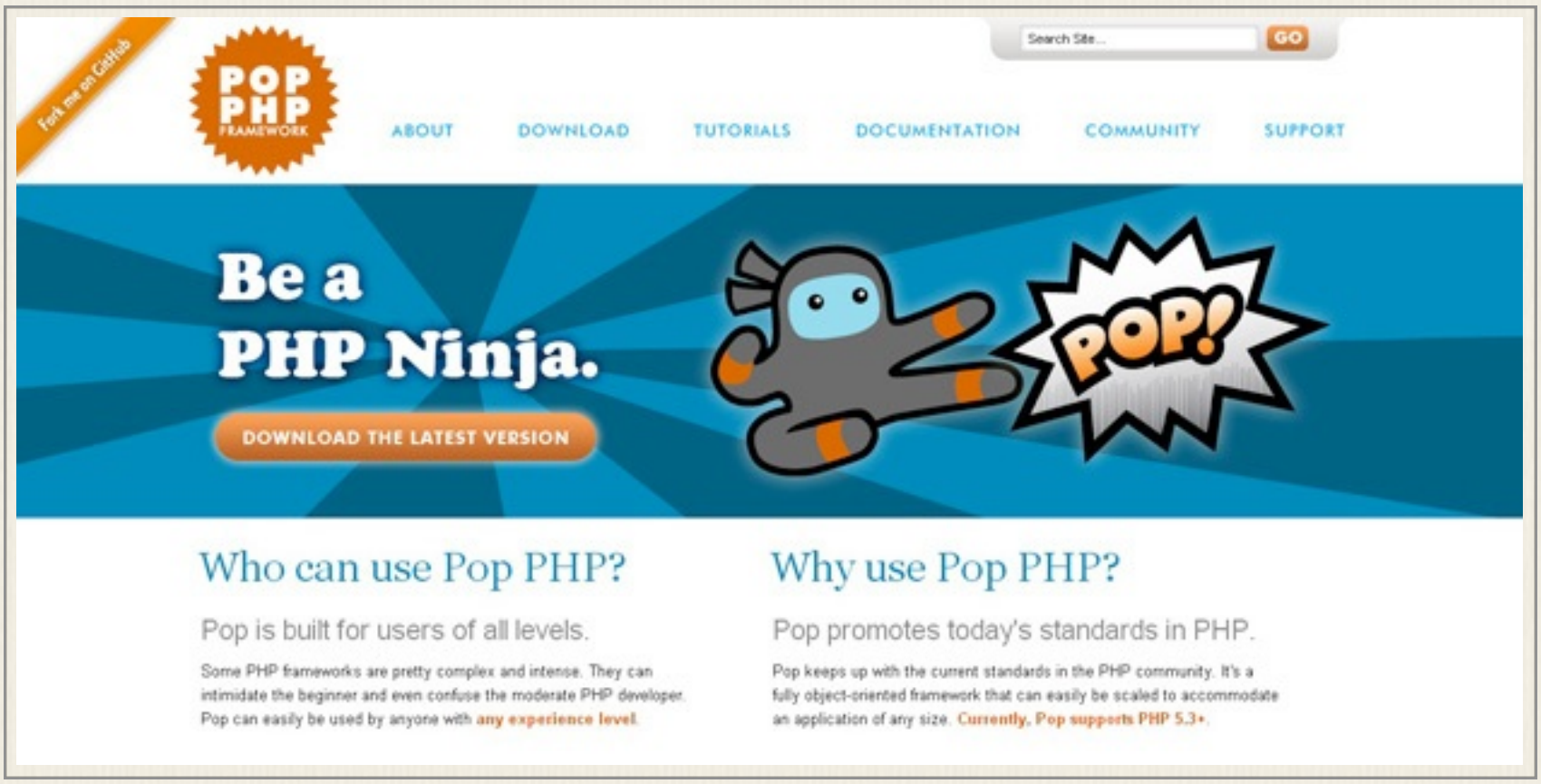
9. PHPPIxie

PHPPixie 是个轻量级的 MVC PHP 框架，能快速设计 web 网站，容易使用并且为 web 开发提供一个坚实的基础。它的设计尽可能的避免了大量的样板和引用减弱网站服务器的加载能力。PHPixie 使用大量的命名规范，所以需要用户配置的地方很少。

10. Kohana

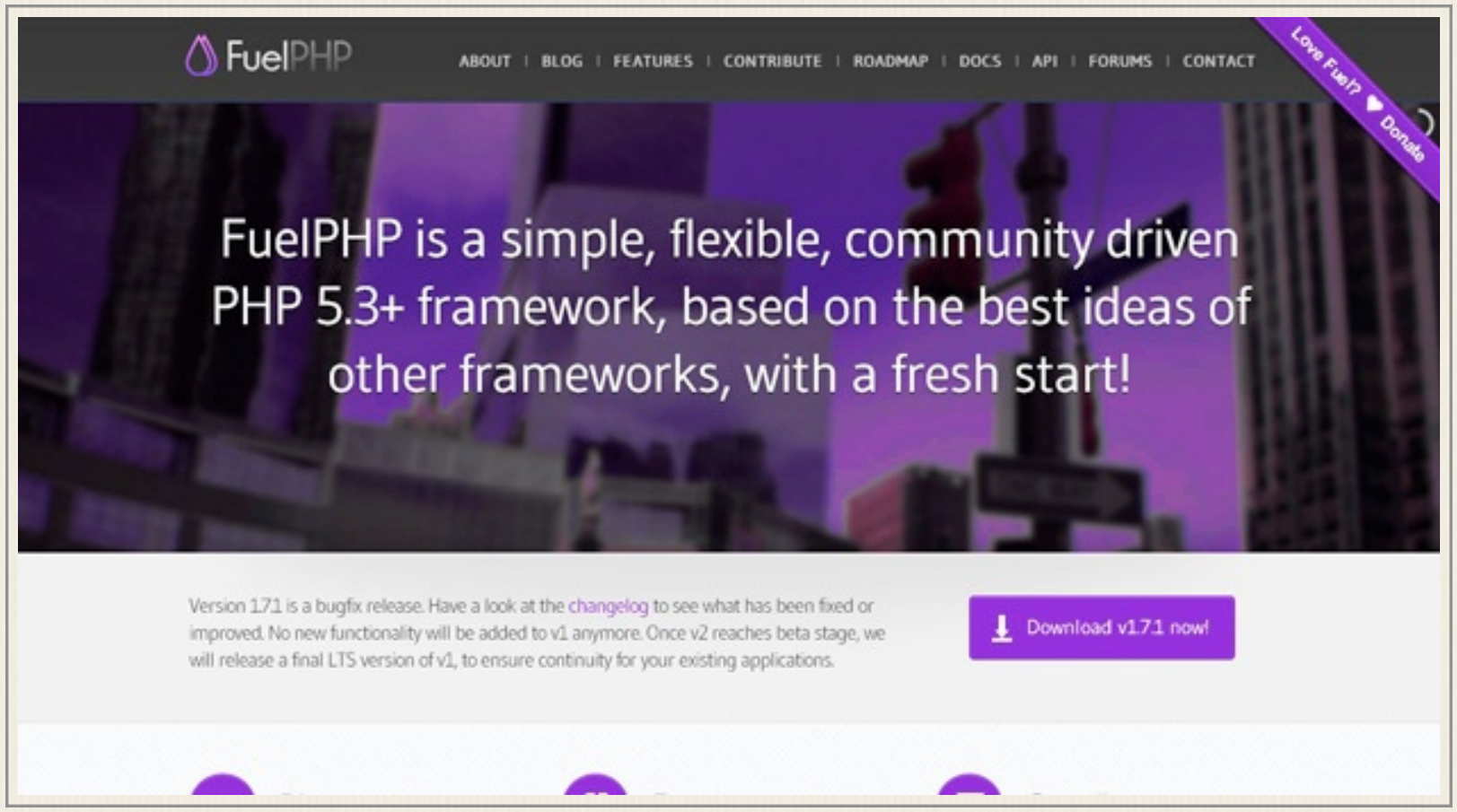
Kohana 是个开源的，面向对象的 MVC web 框架，使用 PHP5 hat 创建的。它提供一系列富组件来创建 web 应用。

11. Pop PHP



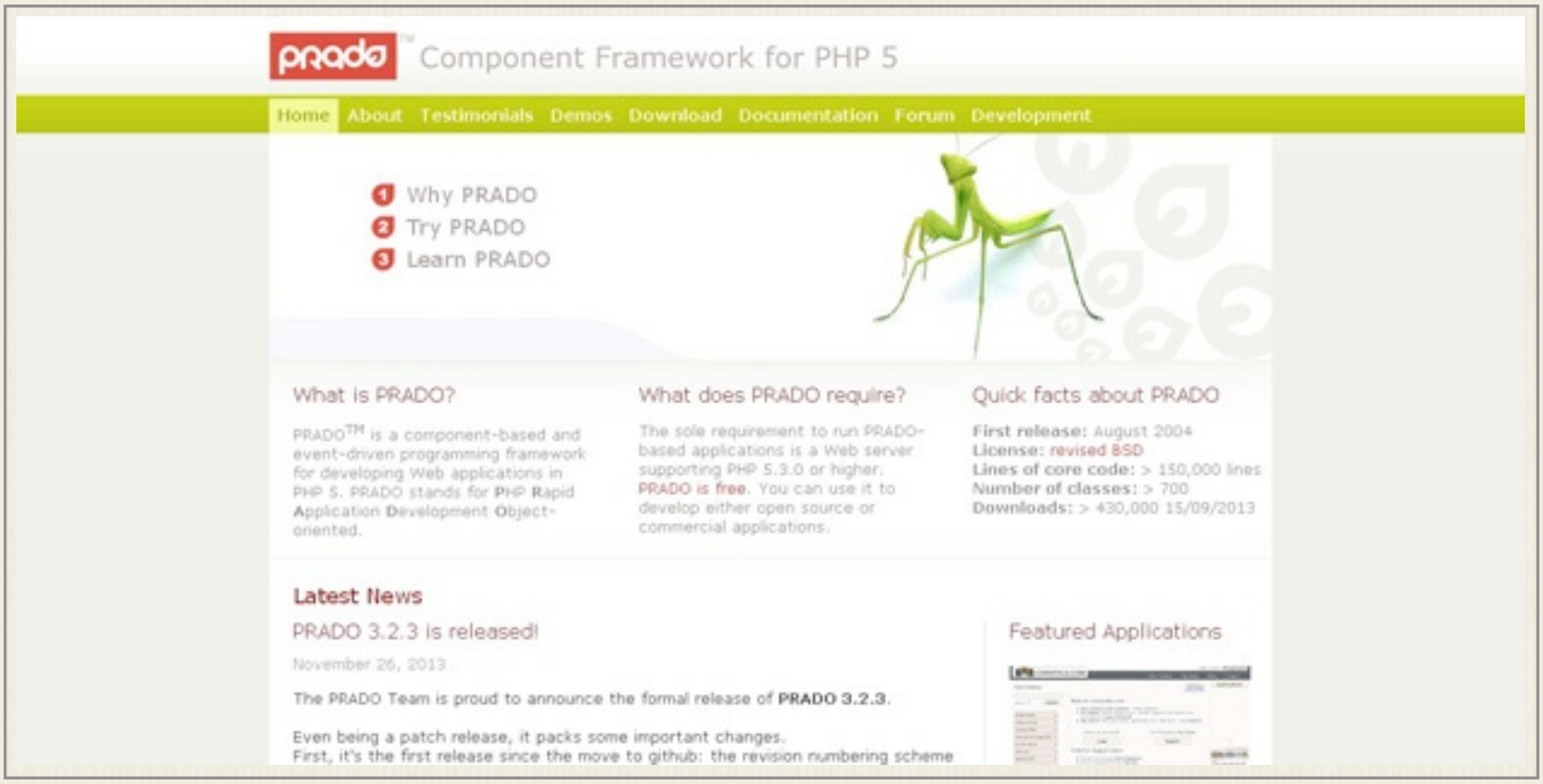
The Pop PHP Framework 是个健壮，但是又易于使用的 PHP 框架，带有一个详细的 API，支持 PHP 5.3+。The Pop PHP 框架虽然功能很强大，但是依然保持着它的简单性和轻量级的特性。即使现在加入了很多新的功能，但是就像工具箱一样易于使用，一直是众多 PHP 框架中的主流框架。

12. FuelPHP



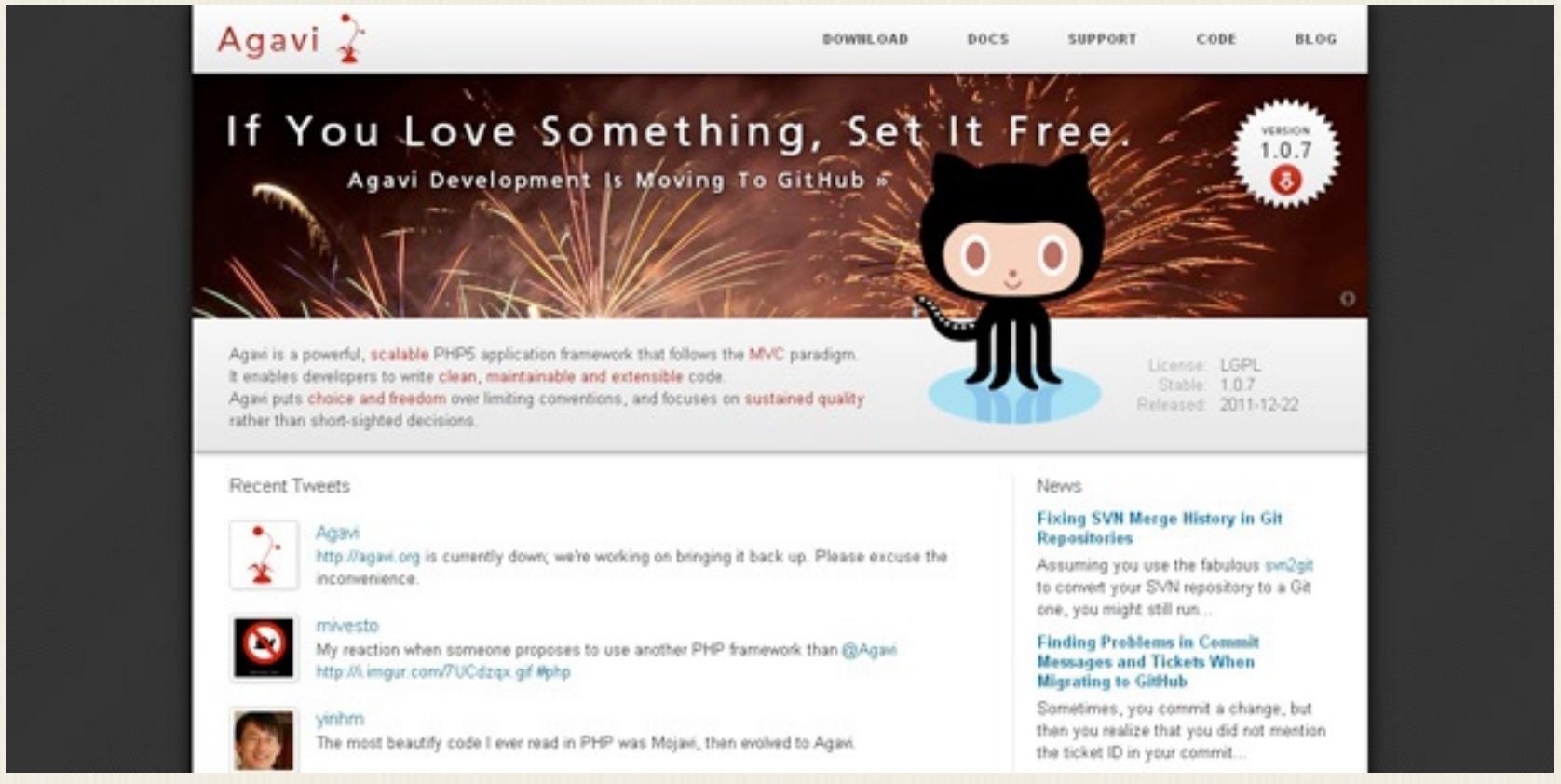
Fuel 是 个简单，灵活，社区驱动的 PHP 5.3 web 框架，基于其他框架的最好想法，是一个新的开始！Fuel 已经在 Apache，IIS 和 Nginx 上测试过，测试结果非常令人满意。Fuel 相比于其他框架提供各种不同的方式去开发，而且努力成为社区驱动的产品。仅仅是接近六个月的时间，就已经有 30 位开发者提交了源代码和文档。

13. Prado



PRADOTM 是个基于组件的，事件驱动的编程框架，用户开发 web 应用，使用 PHP 5 编写。PRADOTM 是 PHP 面向对象快速开发的典型代表。

14. Agavi



Agavi 是个功能强大，可伸缩的 PHP5 应用框架，遵从 MVC paradigm。它能帮助开发者编写简洁，可维护可扩展的代码。它没有太多的自由约束，给用户较高的选择权和自定义的权力，专注于长远的发展，不为一时的利益而改变初衷。

15. DooPhp

DooPHP 是个快速开发的 PHP 框架，使用众所周知的设计模式，比如 MVC 和 ORM。它减少了开发成本，减少开发者的工作量。

原译文链接：http://www.oschina.net/news/48982/best-php-frameworks-2014

原文链接：http://codegeekz.com/best-php-frameworks-2014/

Web前端构建工具版本号管理方案思考

作者：小网

前端构建工具满天飞的情景下，笔者也忍不住去捣鼓了一下，真正体验一下NODEJS带来的魅力，经过一段时间规划设计，终于将平台工具捣鼓出来了。 在里面也体验了express, socket.io, grunt等node插件服务，使用很流畅，并且很好的完了我的基本需求（JS\CSS\IMAGE的压缩和自动部署功能）。

虽然基本功能完成，但是还有一个让人容易忽略而又重要的问题来了，就是资源文件的版本号的问题。在这里用的是SeaJS来做的模块管理，也在网上搜集了一下关于这块的资料。大致有以下两种方案：

1、 配置SeaJS的map对象。很方便，集中管理，但是也存在一些问题，比如：map对象的维护、配置文件引用页面的时间戳问题。

2、 生成文件sign，替换原有文件的名。这里不用考虑页面引用的时间戳问题，但是需要对资源及页面做一次全文查找替换，这对于分散部署的情况下将会非常复杂。

看了上面两种方案后，感觉在我的场景下还存在一些不足之处。笔者想要的是不影响开发情况下，开发人员完全不用考虑文件被缓存问题，只需完成编码，提交到构建系统后，一键完成构建和部署。

确定好目标后，既然不想让开发同学关心时间戳问题，那么是否可以将这个工作交给Web服务器来做呢？那么交给服务器来做之后，如何动态的更新最新的资源文件呢?

有问题就好解决，将问题抛出来之后，方案也渐进呈现出来，不啰嗦了，直接上笔者的实现方案：

构建工具动态生成.htaccess文件，将构建的资源文件的URL重写列表同步到.htaccess文件中以达到由服务器动态获取新的资源文件。

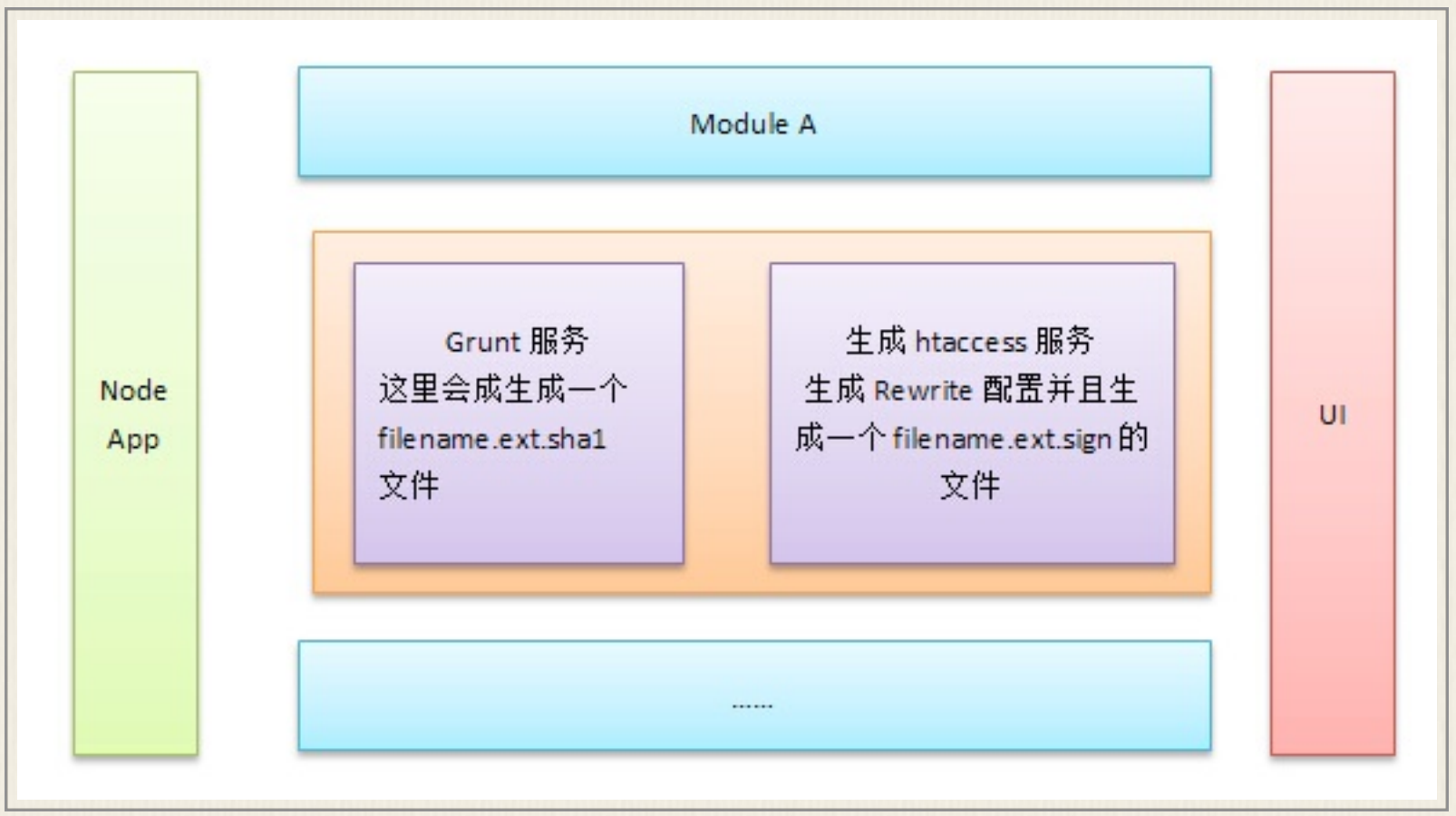
（备注：笔者是在apache环境下，对IIS\NGINX\TOMCAT\RESIN\JBOSS等环境还需要再研究研究。）

看了上面的方案是不是很简单，那具体是怎么做的呢？笔者也在这里列一下。

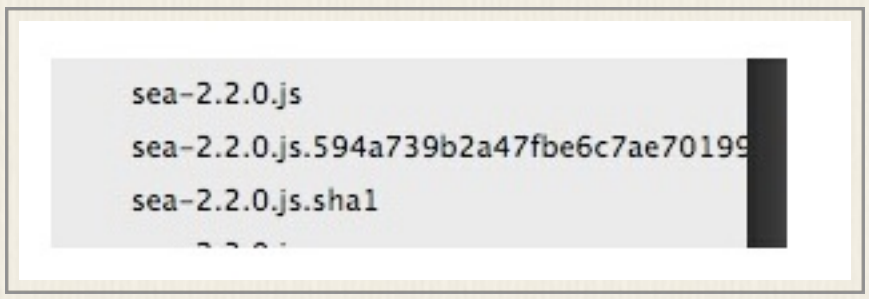
1. 构建工具在构的时候生成一个与filename.ext对应的filename.ext.sha1的文件，这个.sha1的文件存放filename.ext文件的sha1值。

2. 在构建完成后，构建工具读取.htaccess的模板，并且遍历得到所有资源文件的列表，生成一个对filename.ext对应的filename.ext.sign的文件

3. 按规则生成RewriteCond和RewriteRule。并将规则数据写到.htaccess文件中并部署到资源文件站点的根据目录。



图一：构建工具基本结构图



图二：构建后的文件列表

```
addRule : function(filename, sign){
    var uri = filename.replace($proj.base + $proj.workspace.env.path, "/");

    // RewriteCond %{REQUEST_URI} ^/css/g.css$ [NC] [AND]
    // RewriteCond %{QUERY_STRING} !^.+$ [NC]
    // RewriteRule css/g.css$ /css/g.css.1c588b41b1520e0fb76a3ad8e4eaca08329a6a1d [PT,L]

    this.rules.push("RewriteCond %{REQUEST_URI} ^" + uri + "$ [NC] [AND]\n");
    this.rules.push("RewriteCond %{QUERY_STRING} !^.+$ [NC]\n")
    this.rules.push("RewriteRule " + uri.substr(1) + "$ " + uri + "." + sign + " [PT,L]\n\n");
},
```

图三：生成一条Rule

```
1   Options -Indexes +FollowSymLinks
2
3   RewriteEngine On
4
5   RewriteCond %{REQUEST_URI} ^/css/g.css$ [NC] [AND]
6   RewriteCond %{QUERY_STRING} !^.+$ [NC]
7   RewriteRule css/g.css$ /css/g.css.1c588b41b1520e0fb76a3ad8e4eaca08329a6a1d [PT,L]
8
```

图四：生成后的.htaccess文件

OK，到这里基本告一段落了，资源版本号解决了。但是对于性能影响这块还需要观察，待有时间再研究。

Git地址：https://github.com/zwlijun/se.build

原文链接：http://tid.tenpay.com/?p=6351

百度是如何使用hadoop的

作者：数控小V

百度作为全球最大的中文搜索引擎公司，提供基于搜索引擎的各种产品，几乎覆盖了中文网络世界中所有的搜索需求,因此，百度对海量数据处理的要求是比 较高的， 要在线下对数据进行分析，还要在规定的时间内处理完并反馈到平台上。百度在互联网领域的平台需求要通过性能较好的云平台进行处理了，Hadoop就是很好 的选择。在百度，Hadoop主要应用于以下几个方面：

日志的存储和统计;

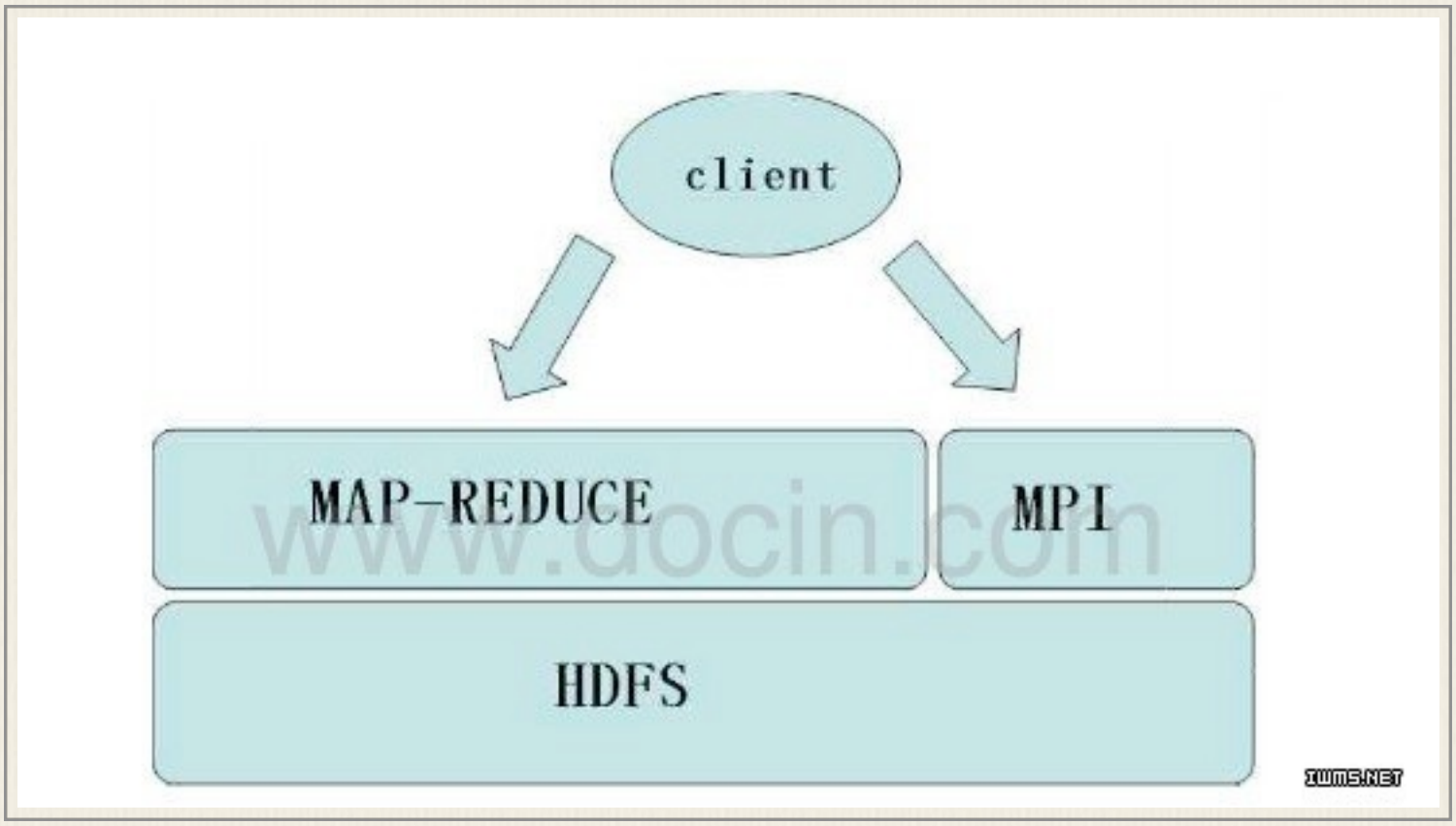
网页数据的分析和挖掘;

商业分析，如用户的行为和广告关注度等;

在线数据的反馈，及时得到在线广告的点击情况;

用户网页的聚类，分析用户的推荐度及用户之间的关联度。

MapReduce主要是一种思想，不能解决所有领域内与计算有关的问题，百度的研究人员认为比较好的模型应该如下图：



HDFS 实现共享存储，一些计算使用MapReduce解决，一些计算使用MPI解决，而还有一些计算需要通过两者来共同处理。因为MapReduce适合处理数 据很大且适合划分的数据，所以在处理这类数据时就可以用MapReduce做一些过滤，得到基本的向量矩阵，然后通过MPI进一步处理后返回结果，只有整 合技术才能更好地解决问题。

百度现在拥有3个Hadoop集群，总规模在700台机器左右，其中有100多台新机器和600多台要淘汰的机器(它们的计算能力相当于200多台新机器)，不过其规模还在不断的增加中。现在每天运行的MapReduce任务在3000个左右，处理数据约120TB/天。

百度为了更好地用Hadoop 进行数据处理，在以下几个方面做了改进和调整:

(1)调整MapReduce策略

限制作业处于运行状态的任务数;

调整预测执行策略，控制预测执行量，一些任务不需要预测执行;

根据节点内存状况进行调度;

平衡中间结果输出，通过压缩处理减少I/O负担。

（2）改进HDFS的效率和功能

权限控制，在PB级数据量的集群上数据应该是共享的，这样分析起来比较容易，但是需要对权限进行限制;

让分区与节点独立，这样，一个分区坏掉后节点上的其他分区还可以正常使用;

修改DSClient选取块副本位置的策略，增加功能使DFSClient选取块时跳过出错的DataNode;

解决VFS(Virtual File System)的POSIX(Portable Operating System Interface of Unix)兼容性问题。

（3）修改Speculative的执行策略

采用速率倒数替代速率，防止数据分布不均时经常不能启动预测执行情况的发生;

增加任务时必须达到某个百分比后才能启动预测执行的限制，解决reduce运行等待map数据的时间问题;

只有一个map或reduce时，可以直接启动预测执行。

（4）对资源使用进行控制

对应用物理内存进行控制。如果内存使用过多会导致操作系统跳过一些任务，百度通过修改Linux内核对进程使用的物理内存进行独立的限制，超过阈值可以终止进程。

分组调度计算资源，实现存储共享、计算独立，在Hadoop中运行的进程是不可抢占的。

在大块文件系统中，X86平台下一个页的大小是4KB。如果页较小，管理的数据就会很多，会增加数据操作的代价并影响计算效率，因此需要增加页的大小。

百度在使用Hadoop时也遇到了一些问题，主要有:

- MapReduce的效率问题：比如，如何在shuffle效率方面减少I/O次数以提高并行效率;如何在排序效率方面设置排序为可配置的，因为排序过程会浪费很多的计算资源，而一些情况下是不需要排序的。

- HDFS的效率和可靠性问题：如何提高随机访问效率，以及数据写入的实时性问题，如果Hadoop每写一条日志就在HDFS上存储一次，效率会很低。

- 内存使 用的问题：reducer端的shuffle会频繁地使用内存，这里采用类似Linux的buddy system来解决，保证Hadoop用最小的开销达到最高的利用率;当Java 进程内容使用内存较多时，可以调整垃圾回收(GC)策略;有时存在大量的内存复制现象，这会消耗大量CPU资源，同时还会导致内存使用峰值极高，这时需要 减少内存的复制。

- 作业调度的问题：如何限制任务的map和reduce计算单元的数量，以确保重要计算可以有足够的计算单元;如何对TaskTracker进行分组控制，以限制作业执行的机器，同时还可以在用户提交任务时确定执行的分组并对分组进行认证。

- 性能提 升的问题：UserLogs cleanup在每次task结束的时候都要查看一下日志，以决定是否清除，这会占用一定的任务资源，可以通过将清理线程从子Java进程移到 TaskTracker来解决;子Java进程会对文本行进行切割而map和reduce进程则会重新切割，这将造成重复处理，这时需要关掉Java进程 的切割功能;在排序的时候也可以实现并行排序来提升性能;实现对数据的异步读写也可以提升性能。

- 健壮性 的问题：需要对mapper和reducer程序的内存消耗进行限制，这就要修改Linux内核，增加其限制进程的物理内存的功能;也可以通过多个map 程序共享一块内存，以一定的代价减少对物理内存的使用;还可以将DataNode和TaskTracker的UGI配置为普通用户并设置账号密码;或者让DataNode和TaskTracker分账号启动，确保HDFS数据的安全性，防止Tracker操作DataNode中的内容;在不能保证用户的每 个程序都很健壮的情况下，有时需要将进程终止掉，但要保证父进程终止后子进程也被终止。

- Streaming 局限性的问题：比如，只能处理文本数据，mapper和reducer按照文本行的协议通信，无法对二进制的数据进行简单处理。为了解决这个问题，百度人 员新写了一个类Bistreaming(Binary Streaming)，这里的子Java进程mapper和reducer按照(KeyLen，Key，ValLen，Value)的方式通信，用户可以 按照这个协议编写程序。

- 用户认证的问题：这个问题的解决办法是让用户名、密码、所属组都在NameNode和Job Track-
er上集中维护，用户连接时需要提供用户名和密码，从而保证数据的安全性。

百度下一步的工作重点可能主要会涉及以下内容：

- 内存方面，降低NameNode的内存使用并研究JVM的内存管理;
- 调度方面，改进任务可以被抢占的情况，同时开发出自己的基于Capacity的作业调度器，让等待作业队列具有优先级且队列中的作业可以设置Capacity，并可以支持TaskTracker分组;
- 压缩算 法，选择较好的方法提高压缩比、减少存储容量，同时选取高效率的算法以进行shuffle数据的压缩和解压;对mapper程序和reducer程序使用 的资源进行控制，防止过度消耗资源导致机器死机。以前是通过修改Linux内核来进行控制的，现在考虑通过在Linux中引入cgroup来对mapper和reducer使用的资源进行控制;将DataNode的并发数据读写方式由多线程改为select方式，以支持大规模并发读写和 Hypertable的应用。

百度同时也在使用Hypertable，它是以Google发布的BigTable为基础的开源分布式数据存储系统，百度将它作为分析用户行为的平台，同时在元数据集中化、内存占用优化、集群安全停机、故障自动恢复等方面做了一些改进。

原文链接：http://www.thebigdata.cn/Hadoop/11335.html

快刀初试：Spark GraphX在淘宝的实践

作者：黄明，吴炜

摘要：由于Spark GraphX性能良好，又有丰富的功能和运算符，能在海量数据上自如运行复杂的图算法，淘宝尝试将它作为分布式图计算平台，进行各种算法尝试和生产应用。本文结合GraphX的原理和特点，分享其在淘宝的应用实践。

早在0.5版本，Spark就带了一个小型的 Bagel模块，提供了类似Pregel的功能。当然，这个版本还非常原始，性能和功能都比较弱，属于实验型产品。到0.8版本时，鉴于业界对分布式图计 算的需求日益见涨，Spark开始独立一个分支Graphx-Branch，作为独立的图计算模块，借鉴GraphLab，开始设计开发GraphX。在 0.9版本中，这个模块被正式集成到主干，虽然是Alpha版本，但已可以试用，小面包圈Bagel告别舞台。1.0版本，GraphX正式投入生产使 用。

值得注意的是，GraphX目前依然处于快速发展中，从0.8的分支到0.9和1.0，每个版本代码都有不少的改进和重 构。根据观察，在没有改任何代码逻辑和运行环境，只是升级版本、切换接口和重新编译的情况下，每个版本有10%~20%的性能提升。虽然和 GraphLab的性能还有一定差距，但凭借Spark整体上的一体化流水线处理，社区热烈的活跃度及快速改进速度，GraphX具有强大的竞争力。

分布式图计算

在 正式介绍GraphX之前，先看看通用的分布式图计算框架。简单来说，分布式图计算框架的目的，是将对于巨型图的各种操作包装为简单的接口，让分布式存 储、并行计算等复杂问题对上层透明，从而使复杂网络和图算法的工程师，更加聚焦在图相关的模型设计和使用上，而不用关心底层的分

布式细节。为了实现该目 的，需要解决两个通用问题：图存储模式和图计算模式。

图存储模式

巨型图的存储总体上有边分割和点分割两种存储方式。2013年，GraphLab2.0将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

边分割：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。

点分割：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量。

虽然两种方法互有利弊，但现在是点分割占上风，各种分布式图计算框架都将自己底层的存储形式变成了点分割。主要原因有以下两个。

- 磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点就类似于常见的空间换时间的策略。
- 在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差非常悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了。

图计算模型

目 前的图计算框架基本上都遵循BSP（Bulk Synchronous Parallell）计算模式。在BSP中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和栅栏同步三个步骤组成。同步完成，标志着这 个超步的完

成及下一个超步的开始。BSP模式很简洁。基于BSP模式，目前有两种比较成熟的图计算模型。

- Pregel模型——像顶点一样思考

2010 年，Google的新的三架马车Caffeine、Pregel、Dremel发布。随着Pregel一起，BSP模型广为人知。Pregel借鉴 MapReduce的思想，提出了“像顶点一样思考”（Think Like A Vertex）的图计算模式，让用户无需考虑并行分布式计算的细节，只需要实现一个顶点更新函数，让框架在遍历顶点时进行调用即可。

常见的代码模板如下：

```
void Compute(MessageIterator* msgs) {
  // 遍历由顶点入边传入的消息列表
  for (; !msgs->Done(); msgs->Next())
    doSomething()
  // 生成新的顶点值
  *MutableVertexValue() = ...
  // 生成沿顶点出边发送的消息
  SendMessageToAllNeighbors(...);
}
```

这个模型虽然简洁，但很容易发现它的缺陷。对于邻居数很多的顶点，它需要处理的消息非常庞大，而且在这个模式下，它们是无法被并发处理的。所以对于符合幂律分布的自然图，这种计算模型下很容易发生假死或者崩溃。

- GAS模型——邻居更新模型

相 比Pregel模型的消息通信范式，GraphLab的GAS模型更偏向共享内存风格。它允许用户的自定义函数访问当前顶点的整个邻域，可抽象成

Gather、Apply和Scatter三个阶段，简称为GAS。相对应，用户需要实现三个独立的函数gather、apply和scatter。常见 的代码模板如下所示：

```
// 从邻居点和边收集数据
Message gather(Vertex u, Edge uv, Vertex v)
{
  Message msg = ...
  return msg
}
// 汇总函数
Message sum(Message left, Message right) {
  return left+right
}
// 更新顶点Master
void apply(Vertex u, Message sum) {
  u.value = ...
}
// 更新邻边和邻居点
void scatter(Vertex u, Edge uv, Vertex v) {
  uv.value = ...
  if ((|u.delta|>ε) Active(v)
}
```

由于gather/scatter函数是以单条边为操作粒度，所以对于一个顶点的众多邻边，可以分别由相应的worker独立调用gather/scatter函数。这一设计主要是为了适应点分割的图存储模式，从而避免Pregel模型会遇到的问题。

GraphX的框架

在 设计GraphX时，点分割和GAS都已成熟，并在设计和编码中针对它们进行了优化，在功能和性能之间寻找最佳的平衡点。如同Spark本身，每个子模块 都有一个核心抽象。GraphX的核心抽象是Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了Spark RDD的抽象，有Table和Graph两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。

如 同Spark，GraphX的代码非常简洁。GraphX的核心代码只有3千多行，而在此之上实现的Pregel模型，只要短短的20多行。GraphX 的代码结构整体如图1所示，其中大部分的实现，都是围绕Partition的优化进行的。这在某种程度上说明了点分割的存储和相应的计算优化，的确是图计 算框架的重点和难点。

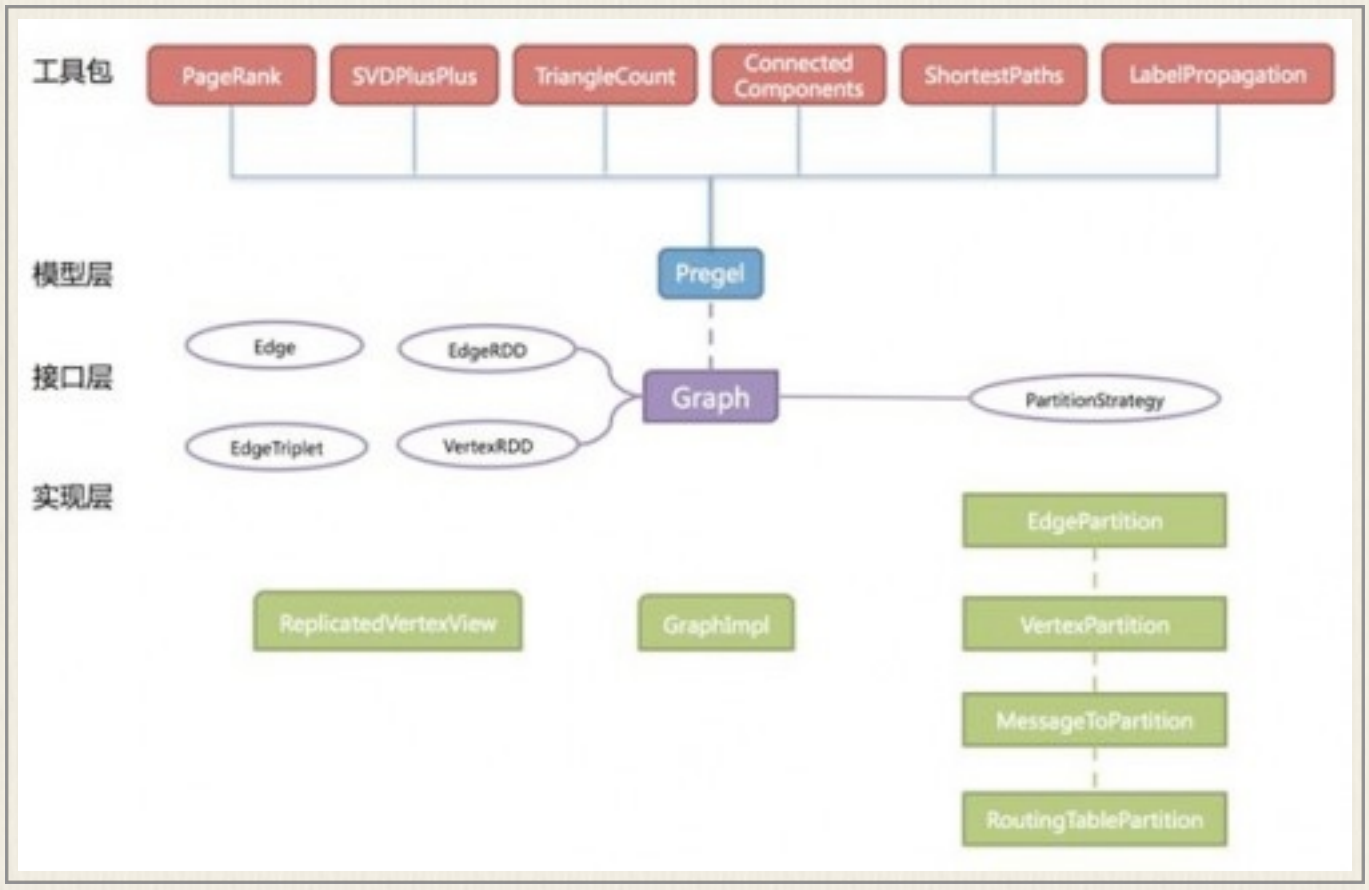


图1 GraphX的代码结构

GraphX的底层设计有以下几个关键点。

1.对 Graph视图的所有操作，最终都会转换成其关联的Table视图的RDD操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列RDD的转换过 程。因此，Graph最终具备了RDD的3个关键特性：Immutable、Distributed和Fault-Tolerant。其中最关键的是 Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX会有一定程度的不变顶点和边的复用优化，对用户透明。

2.两 种视图底层共用的物理数据，由RDD[Vertex-Partition]和RDD[EdgePartition]这两个RDD组成。点和边实际都不是以 表Collection[tuple]的形式存储的，而是由VertexPartition/EdgePartition在内部存储一个带索引结构的分片 数据块，以加速不同视图下的遍历速度。不变的索引结构在RDD转换过程中是共用的，降低了计算和存储开销。

3.图 的分布式存储采用点分割模式，而且使用partitionBy方法，由用户指定不同的划分策略（PartitionStrategy）。划分策略会将边分 配到各个EdgePartition，顶点Master分配到各个VertexPartition，EdgePartition也会缓存本地边关联点的 Ghost副本。划分策略的不同会影响到所需要缓存的Ghost副本数量，以及每个EdgePartition分配的边的均衡程度，需要根据图的结构特征 选取最佳策略。目前有EdgePartition2d、EdgePartition1d、RandomVertexCut和 CanonicalRandomVertexCut这四种策略。在淘宝大部分场景下，EdgePartition2d效果最好。

GraphX的图运算符

如同Spark一样，GraphX的Graph类提供了丰富的图运算符，大致结构如图2所示。可以在官方GraphX Programming Guide中找到每个函数的详细说明，本文仅讲述几个需要注意的方法。

图2 GraphX的图运算符

图的cache

每 个图是由3个RDD组成，所以会占用更多的内存。相应图的cache、unpersist和checkpoint，更需要注意使用技巧。出于最大限度复用 边的理念，GraphX的默认接口只提供了unpersistVertices方法。如果要释放边，调用g.edges.unpersist()方法才 行，这给用户带来了一定的不便，但为GraphX的优化提供了便利和空间。参考GraphX的Pregel代码，对一个大图，目前最佳的实践是：

```
var g=...
var prevG: Graph[VD, ED] = null
while(...){
  prevG = g
  g = doSomething(g)
  g.cache()
  prevG.unpersistVertices(blocking=false)
  prevG.edges.unpersist(blocking=false)
}
```

大体之意是根据GraphX中Graph的不变性，对g做操作并赋回给g之后，g已不是原来的g了，而且会在下一轮迭代使用，所以必须cache。 另外，必须先用prevG保留住对原来图的引用，并在新图产生后，快速将旧图彻底释放掉。否则，十几轮迭代后，会有内存泄漏问题，很快耗光作业缓存空间。

mrTriplets——邻边聚合

mrTriplets（mapReduceTriplets）是GraphX中最核心的一个接口。Pregel也基于它而来，所以对它的优化能很大程度上影响整个GraphX的性能。mrTriplets运算符的简化定义是：

```
def mapReduceTriplets[A](
  map: EdgeTriplet[VD, ED] =>
Iterator[(VertexID, A)],
  reduce: (A, A) => A)
  : VertexRDD[A]
```

它的计算过程为：map，应用于每一个Triplet上，生成一个或者多个消息，消息以Triplet关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每一个Vertex上，将发送给每一个顶点的消息合并起来。

mrTriplets最后返回的是一个VertexRDD[A]，包含每一个顶点聚合之后的消息（类型为A），没有接收到消息的顶点不会包含在返回的VertexRDD中。

在最近的版本中，GraphX针对它进行了一些优化，对于Pregel以及所有上层算法工具包的性能都有重大影响。主要包括以下几点。

- Caching for Iterative mrTriplets & Incremental Updates for Iterative mrTriplets：在很多图分析算法中，不同点的收敛速度变化很大。在迭代后期，只有很少的点会有更新。因此，对于没有更新的点，下一次mrTriplets计算时EdgeRDD无需更新相应点值的本地缓存，大幅降低了通信开销。
- Indexing Active Edges：没有更新的顶点在下一轮迭代时不需要向邻居重新发送消息。因此，mrTriplets遍历边时，如果一条边的邻居点值在上一轮迭代时没有更新，则直接跳过，避免了大量无用的计算和通信。
- Join Elimination：Triplet 是由一条边和其两个邻居点组成的三元组，操作Triplet的map函数常常只需访问其两个邻居点值中的一个。例如，在PageRank计算中，一个点值 的更新只与其源顶点的值有关，而与其所指向的目的顶点的值无关。那么在mrTriplets计算中，就不需要VertexRDD和EdgeRDD的3- way join，而只需要2-way join。

所有这些优化使GraphX的性能逐渐逼近GraphLab。虽然还有一定差距，但一体化的流水线服务和丰富的编程接口，可以弥补性能的微小差距。

进化的Pregel模型

GraphX中的Pregel接口，并不严格遵循Pregel模型，它是一个参考GAS改进的Pregel模型。定义如下：

```
def pregel[A](initialMsg: A, maxIterations:
  Int, activeDirection: EdgeDirection)(
  vprog: (VertexID, VD, A) => VD,
  sendMsg: EdgeTriplet[VD, ED] =>
Iterator[(VertexID,A)],
  mergeMsg: (A, A) => A)
  : Graph[VD, ED]
```

这种基于mrTrilets方法的Pregel模型，与标准Pregel的最大区别是，它的第2段参数体接收的是3个函数参数，而不接收 messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个Ghost副本收到的消息聚合后，发送给Master副本，再使用vprog 函数来更新点值。消息的接收和发送都被自动并行化处理，无需担心超级节点的问题。

常见的代码模板如下所示：

```
// 更新顶点
vprog(vid: Long, vert: Vertex, msg: Double):
Vertex = {
  v.score = msg + (1 - ALPHA) * v.weight
}
// 发送消息
sendMsg(edgeTriplet: EdgeTriplet[…]):
Iterator[(Long, Double)]
  (destId, ALPHA * edgeTriplet.srcAttr.
score * edgeTriplet.attr.weight)
}
// 合并消息
mergeMsg(v1: Double, v2: Double): Double = {
  v1+v2
}
```

可以看到，GraphX设计这个模型的用意。它综合了Pregel和GAS两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式， 胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的Pregel

版本是最简单的一个版本。对于复杂的业务场景，根据这个版本扩展一个定制的 Pregel是很常见的做法。

图算法工具包

GraphX也提供了一套图算法工具包， 方便用户对图进行分析。目前最新版本已支持PageRank、数三角形、最大连通图和最短路径等6种经典的图算法。这些算法的代码实现，目的和重点在于通 用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。另外，研读这些代码，也是理解GraphX编程最佳实践的好方法。

GraphX在淘宝

图谱体检平台

基 本上，所有的关系都可以从图的角度来看待和处理，但到底一个关系的价值多大？健康与否？适合用于什么场景？很多时候是靠运营人员和产品经理凭直觉来判断和 评估的。如何将各种图的指标精细化和规范化，对于产品和运营的构思进行数据上的预研指导，提供科学决策的依据，是图谱体检平台设计的初衷和出发点。

基于这样的出发点，借助GraphX丰富的接口和工具包，针对淘宝内部林林总总的图业务需求，我们开发一个图谱体检平台。目前主要进行下列指标的检查。

度分布：这 是一个图最基础和重要的指标。度分布检测的目的，主要是了解图中“超级节点”的个数和规模，以及所有节点度的分布曲线。超级节点的存在对各种传播算法都会 有重大的影响（不论是正面助力还是反面阻力），因此要预先对这些数据量有个预估。借助GraphX最基本的图信息接口degrees: VertexRDD[Int]（包括inDegrees和outDegrees），这个指标可以轻松计算出来，并进行各种各样的统计。

二跳邻居数：对 大部分社交关系来说，只获得一跳的度分布远远不够，另一个重要的指标是二跳邻居数。例如，无秘App中好友的好友的秘密，传播范围更广，信息量更丰富。因 此，二跳邻居数的统计是图谱体检中很重要的一个指标。对于二跳邻居的计算，GraphX没有给出现成的接口，需要

自己设计和开发。目前使用的方法是：第一 次遍历，所有点向邻居点传播一个带自身ID，生命值为2的消息；第二次遍历，所有点将收到的消息向邻居点再转发一次，生命值为1；最终统计所有点上，接收 到的生命值为1的ID，并进行分组汇总，得到所有点的二跳邻居。

值得注意的是，进行这个计算之前，需要借助度分布将图中的超级节点去掉，不纳入二跳邻居数的计算。否则，这些超级节点会在第一轮传播后收到过多的消息而爆掉，同时它们参与计算，会影响与它们有一跳邻居关系的顶点，导致不能得到真正有效的二跳邻居数。

连通图：检 测连通图的目的是弄清一个图有几个连通部分及每个连通部分有多少顶点。这样可以将一个大图分割为多个小图，并去掉零碎的连通部分，从而可以在多个小子图上 进行更加精细的操作。目前，GraphX提供了ConnectedComponents和StronglyConnected-Components算 法，使用它们可以快速计算出相应的连通图。连通图可以进一步演化变成社区发现算法，而该算法优劣的评判标准之一，是计算模块的Q值，来查看所谓的modularity情况。

更多的指标，例如Triangle Count和K-Core，无论是借助GraphX已有的函数，还是自己从头开发，都陆续在进行中。目前这个图谱体检平台已初具规模，通过平台的建立和推 广，图相关的产品和业务逐渐走上“无数据，不讨论，用指标来预估效果”的数据化运营之路，有效提高沟通效率，为各种图相关的业务开发走上科学化和系统化之 路做好准备。

多图合并工具

在图谱体检平台的基础上，我们可以了解到各种关系的特点。每种关系都有自己的强项和弱项，例如 有些关系图谱连通性好些，而有些关系图谱的社交性好些，因此往往要使用关系A来丰富关系B。为此，借助GraphX，我们在图谱体检平台上开发了一个多图 合并工具，提供类似图并集的概念，可快速对指定的两个不同的关系图谱进行合并，产生一个新的关系图谱。

以用基于A关系的图来扩充基于B关系 的图，生成扩充图C为例，融合算法的基本思路如图3：若图B中某边的两个顶点都在图A中，则将该边加入C图（如BD边）；若图B中某边的一个顶点在图A 中，另一个顶点不在，则将

该边和另一顶点都加上（如CE边和E点）；若图A中某边的两个顶点都不在图B中，则舍弃这条边和顶点（如EF边）。

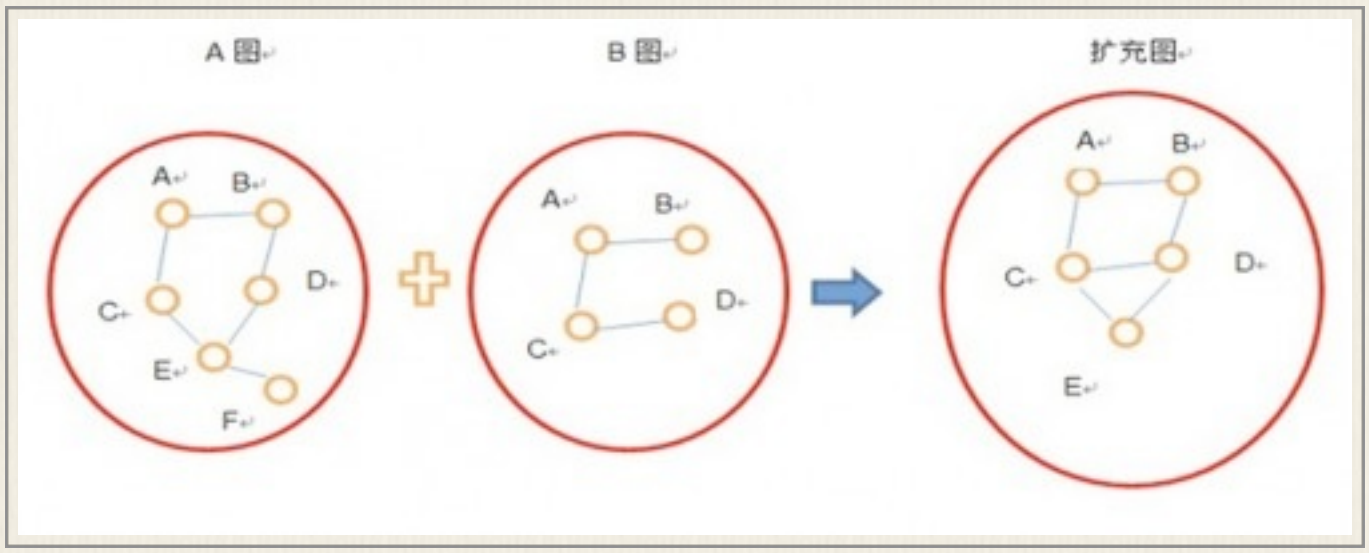


图3 图合并算法

使用 GraphX的outerJoinVertices等图运算符，能很简单地完成上述操作。另外，在考虑图合并时，也可为不同图的边加上不同的权重，综合考虑点之间不同关系的重要性。新产生的图，会再进行一轮图谱体检。通过前后三个图各个体检指标的对比，有助于对业务上线后的效果做预估和判断。如果不符合期 望，可尝试重新选择扩充方案。

能量传播模型

加权网络上的能量传播是经典的图模型之一，可用于用户信誉度预 测。模型的思路是：物以类聚，人以群分。常和信誉度高的用户进行交易的，信誉度自然较高；常和信誉度差的用户有业务来往的，信誉度自然较低。模型不复杂， 但淘宝全网有上亿的用户点和几十亿关系边，要对如此规模的巨型图进行能量传播，并对边的权重进行精细的调节，对图计算框架的性能和功能都是巨大的考验。借 助GraphX，我们在这两点之间取得了平衡，成功实现了该模型。

流程如图4所示，先生成以用户为点、买卖关系为边的巨型图 init-Graph，对选出的种子用户，分别赋予相同的初始正负能量值（trustRank & badRank），然后进行两轮随机游走，一轮好种子传播正能量（tr），一轮坏种子传播负能量（br），然后正负能量相减得到finalRank，根据finalRank判断用户的好坏。边的初始传播强度是0.85，这时AUC很低，需要再给每条边，带上一个由多个特征（交易次数、金额……）组成的组合

权重。每个特征，都有不同的独立权重和偏移量。通过使用partialDerivativeAUC方法，在训练集上计算AUC，然后对AUC求偏导，得到 每个关系维度的独立权重和偏移量，生成新的权重调节器（WeightAdjustor），对图上所有边上的权重更新，再进行新一轮大迭代，这样一直到 AUC稳定时，终止计算。

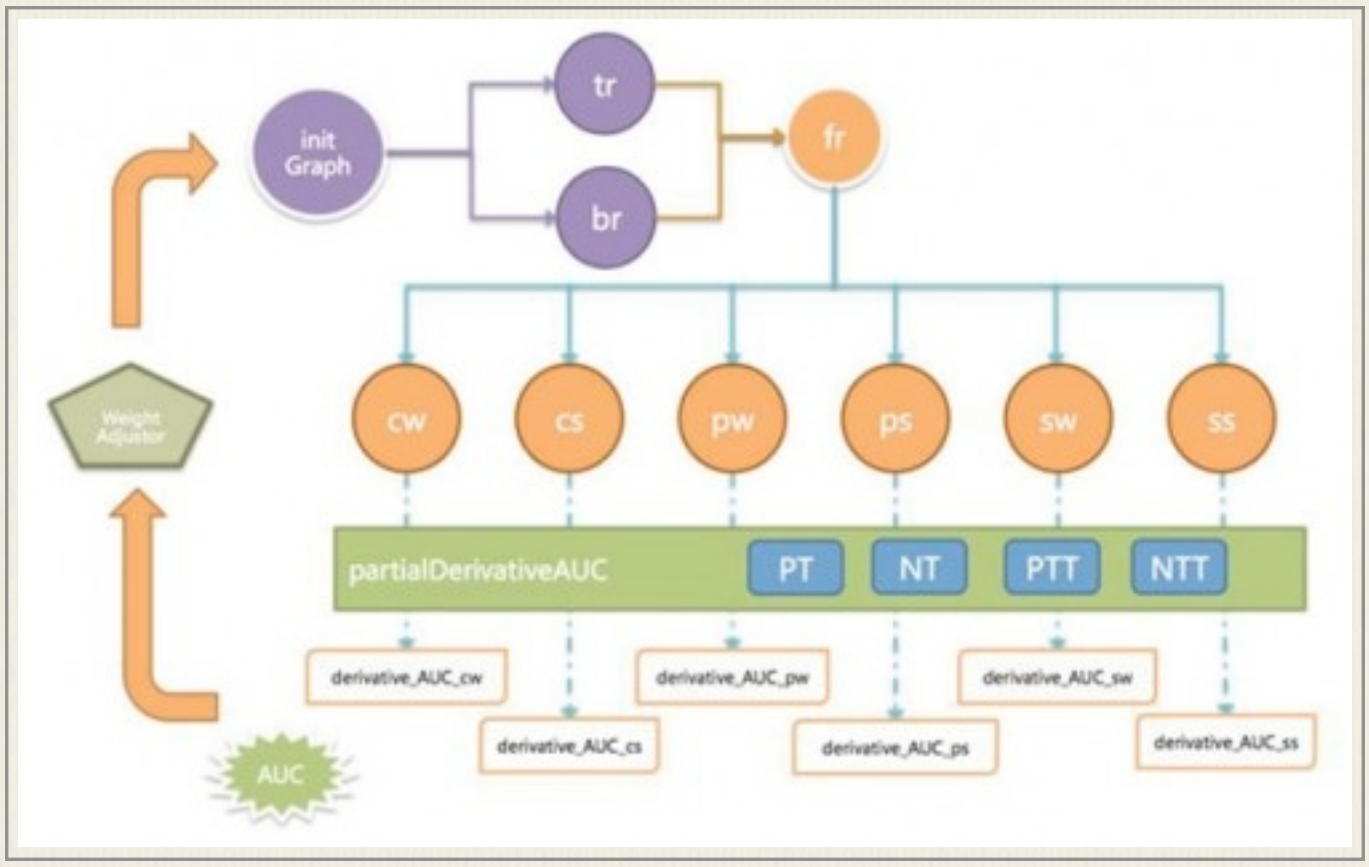


图4 能量传播和训练模型

在接近全量的数据上进行3轮大迭代，每轮2+6次Pregel，每次Pregel大约30次小迭代后，最终的AUC从0.6提升到0.9，达到了不错的用户预测准确率。训练时长在6个小时左右，无论在性能还是准确率上，都超越业务方的期望。

未来图计算的前景

经 过半年多的尝试，之前一些想做但因为没有足够的计算能力而不能实现的图模型，现已不是问题。我们将会进一步将越来越多的图模型，在GraphX上实现。这 些模型应用于用户网络的社区发现、用户影响力、能量传播、标签传播等，可以提升用户黏性和活跃度；而应用到推荐领域的标签

推理、人群划分、年龄段预测、商 品交易时序跳转，则可以提升推荐的丰富度和准确性。

原文链接：http://www.csdn.net/article/2014-08-07/2821097

关于推荐系统中的特征工程

作者：Reprehensible

在多数数据和机器学习的blog里，特征工程 Feature Engineering 都很少被提到。做模型的或者搞Kaggle比赛的人认为这些搞feature工作繁琐又不重要不如多堆几个模型，想入手实际问题的小朋友又不知道怎么提取feature来建模型。我就用个性化推荐系统做个例子，简单说说特征工程在实际的问题里是怎么做。

定义

特征工程 Feature Engineering 在一篇Kaggle blog http://blog.kaggle.com/2014/08/01/learning-from-the-best/ 上有很好的定义：

By feature engineering, I mean using domain specific knowledge or automatic methods for generating, extracting, removing or altering features in the data set.

基本上说是，用目标问题所在的特定领域知识或者自动化的方法来生成、提取、删减或者组合变化得到特征。这些特征可能是显而易见比如说商品的品牌，也 有可能需要复杂的模型计算，比如Facebook上用户A和用户B之间关系的紧密程度（FB使用了一个决策树来生成一个描述这个程度的向量，这个向量决定 了他们News Feed推荐内容。）。这篇blog覆盖了一些用领域知识的方法，自动化方法在这里没有提及。

背景

个性化推荐系统 Personalized recommender system 比其他的机器学习方法更依赖特征工程，所以我拿它来当作问题的背景，结合我之前做过的一个推荐系统里相关经验来说说特征工程具体是个什么东西。

关于推荐系统和个性化推荐系统，可以参看wikipediahttp://en.wikipedia.org/wiki/Recommender_system 具 体不赘述，以下的要点也尽量点到为止，否则这篇又成了“收藏了Mark了”但是不会读的冗余长文。以下如果不特别指出，我就用推荐系统作为“个性化商品推荐系统”的简称。简单来说，推荐系统就是你买了商品A，我们给你算出来个推荐列表 B C D E 等等。商业上来说个性化的推荐比一般化的推荐更能吸引顾客点击或购买，所以利用特征功能提取这些“个性化”的特征放到推荐模型里就很重要，比如在我们的推 荐系统里，把“品牌”的特征加进去，相对于baseline 提高了20%左右的nDCG。推荐系统可以是机器学习的模型也可以是基于关联或者统计规则的模型，对后者来说特征对推荐效果的提升占的比重更大。

利用领域知识生成和提取特征

这几乎是特征工程里占大半时间的工作了：如何描述个性化并且用变量表示成特征。一般方法就是，想想你就是该商品的目标用户，你会想要什么样的个性化。

比如说我们做一个女性衣服类的推荐引擎，这个网站卖各种牌子颜色尺寸等。我们列出可能相关的一些特征，然后在实际购买数据里面检查他们是否对购买结 果产生影响和关联性。比如，从购买数据里可以看出，女性对衣服的品牌多数有固定偏好，比如我太太就很喜欢LOFT的衣服。这些能对购买产生影响的因素都可 以成为特征。

这部分工作需要很多领域知识，一般需要一组的研究人员讨论，要认真的思考这个特定问题有些什么和别的问题不同的特征，也建议和市场部销售部等有领域 知识的专家讨论。经验上来说，这些特征提取的越多越好，并不用担心特征过多，因为推荐系统的数据量都比较大，并且基于一些规则可以很好的筛选特征。

很多机器学习的方法也可以拿来提取一些比较不容易得到的直接特征，比如说原始数据里面没有人工标记过商品的颜色，这些颜色可以通过图像识别得到。统计规则也可以从销售数据里得到一些特征，比如该商品的流行程度。

注意，这些特征可能是固定不变的，比如颜色，品牌等。它也有可能随着时间变化，比如商品的销售排名。实际经验来说，时间变化采样的颗粒度

要按照实际推荐效果来决定，很可能过去三个月的销售排名对推荐效果来说可以很稳定，也或许昨天的排名对今天的推荐效果比三个月平均更好。

特征的表达

大家都知道特征可以是“红”“绿”“蓝”这些离散特征，也可以是1.57这样的连续值特征。一个特征具体如何表达，要看在它在具体模型上怎么用。某些特定问题更倾向于离散特征，因为像推荐系统这样数据很大的情况可以利用模型训练这些特征得到比连续值表达更好的效果。

比如说，商品的流行度可以是一个特征，因为对于某些流行的商品大家都抢着买，喜欢跟风买热门商品这一特性可以作为推荐的特征。我们可以按照销量排名 然后归一化得到每个商品的流行度值，但是直接用这个连续值会有一些问题，比如说用户甲买了流行度分数为 0.75 0.5 0.2 0.1的四个商品，用户乙买了流行度为0.7的一个商品，他们两个怎么比?

如果还记得算法书上说的，定义几个桶buckets，把流行度分到这几个buckets里面，可以解决这个问题。比如定义三个桶：很流行1-0。 95，较流行0.95-0.75，普通0.75-0.4。这样用户甲的特征就是［0，1，1］用户乙的特征就是［0，1，0］，这样你的推荐模型就可以做 一些对比他们俩的相似度或者其他推荐计算了。

顺道提一下就是，为什么在这里直接把0.4之后的丢掉了以及为什么取了三个buckets。这个要看具体问题里面具体特征的用处。

1. 如果这个模型是学习训练出来的，可以用一些feature selection的办法自动去掉一些不需要的bucket。对于那些不是学习出来的模型比如是简单的相似性模型，按照实际推荐效果思考一下用户的行为特 征，需要丢弃一些特征。我之前包含过0.4以下的部分但是实际测试的时候发现推荐结果会恶化，也就是说对于我们的问题，用户喜欢跟风买热门的，但是不喜欢一直买冷门的。

2. 特征需要按照实际购买数据进行修正和理解。三个buckets是我们系统里效果最好的。

一个比较高级的例子是 Facebook 在他们的 Machine Learning meetup 上提到的推荐News feed的特征。每个用户对于其他用户的 news 的点赞和留言以及其他的动作都会得到一个评价值，这是一系列的连续值，直接拿来

训练模型效果不好。他们的做法是做了一个简单的决策树，训练的输入是这些连 续值，训练目标是看对于用户A是否应该显示用户B的 news 。这个决策树显然很粗糙，但是树的每个叶子节点可以成为一个特征，那么这些叶子节点就可以当作用户A的特征向量，拿来训练其他模型比如Logistic Regression，效果不错。

直接特征和间接特征

直接特征 Extacted Feature 就是比如商品的品牌，间接特征 Derived Feature 可以是从直接特征或者各种数据组合里计算推导出来的。

间接特征的一个例子还是品牌特征，拿女性服饰类举例。比如我太太很喜欢LOFT的衣服，但如果一个推荐引擎使劲给推荐LOFT牌的衣服，她也会很 烦。所以品牌并不完全是一个直接特征，它可以有一些变化。比如从购买数据里面看到，购买了LOFT牌衣服的，有20%也购买了J Crew牌，15%也购买了Ann Taylor牌。所以 LOFT 这个特征应该变成一个向量 [LOFT:1, J_Crew:0.2, Ann_Taylor:0.15, …]。实际效果上它提高了推荐的多样性，在多个测试函数中都有不错的提升。

间接特征另外一个高级一些的例子就是用户职业。绝大多数用户都不会填自己的职业等个人相关信息，主要是因为隐私或者就是因为麻烦。从用户的购买记录 和浏览记录里面，我们可以定义几个预设的职业类型然后用户的职业预测到这几个类型里。比如用户买过一些转换插头和充电器还有旅行电脑包，所以他可能常外出 旅行，所以以后推荐的商品可能是轻便携带；又比如用户买过母婴用品就知道该用户可能自己是妈妈或者家里有小孩。

关于间接特征我印象最深的是美国亚马逊上的性别特征。我有一次给我太太买了给女生修眉毛的剃刀，亚马逊的推荐内容就立刻从推荐相机鼠标键盘等男性特征较强的变成了推荐时尚杂志这种有女性特征的东西。

间接特征的提取可以用到很多机器学习的技术，比如根据商品的文本描述提取它的文本向量，以这些文本向量为特征训练多分类的分类器，可以把商品分类对 应到各种用户职业特征上。它也可以利用人工标记的类型列表，比如时尚杂志的女性特征。它也可以利用一些统计规则，比如单反相机的购买记录里，男性的比例会 高于女性，所以单反相机的性别特征向量可以是购买人数性别的比例值。

特征选择

这部分的工作就看起来比较高级一些，比较贴近机器学习的研究工作。一般来说是两个方法：基于领域知识的手工选择以及自动选择方法。

对于关联规则和统计规则的模型来说，手工选择的比重要大一些。比如我们已有了baseline的特征向量，现在加进去品牌偏好，给一定的权值，看评 价函数输出的结果是否增强了推荐效果。对于学习的模型来说，可以通过模型自动选择每个特征的权值，按照和效果的关联来调整模型的参数。这里需要提醒的是， 这个选择过程不是单增单减，很可能遇到两个特征组合A＋B效果很好，A＋C效果也很好，但是A＋B＋C效果就呵呵了。个人建议在关联和统计规则里面把最重 要的几个特征放进去然后优化关于这几个特征的规则，把复杂的特征选择留给学习出来的模型。

自动选择方法就很多了，用的也是常用的各种自动选择方法，什么forward selection啊backward selection啊各种regularization等等，全写在这里篇幅就太长，推荐看看一些其他关于机器学习里 feature selection 的blog和综述。值得提醒的是选择方法和评价函数的关联。推荐系统的评价函数一般不是AUC曲线等按照error计算的函数，也就是说推荐的效果并不是 按照“精准”来衡量，要遵循特定问题需要的评价函数比如nDCG，所以以error matrix为标准的一些方法可能效果会不好。比如说用PCA降低特征的维度很可能把那些对推荐效果很重要的长尾特征给舍弃了。

特征的组合变化

这部分工作看起来就比较碉堡，可发挥的空间就看你的想像力和经验了。这里的组合变化远不限于把已有的特征加减乘除（比如Kernel Tricks之类），我举个比较有想像力的例子。

现在市面上社交网络里面“你可能认识的人”的推荐算法几乎都是基于补全网络的办法，这样推荐的人可能只是单纯的补充和完善朋友圈，推荐的人可能很无 趣，这样会导致推荐效果不好让用户失去接受推荐的兴趣。目测新浪微博用的还是这种补全的方法，因为整天向我推荐丁一晨李开复姚晨等人或者最近关注的人的共 同关注人，所以推荐的人很无趣几乎都不会点关注。

斯坦福小帅哥教授 Jure Leskovec 在2010年的一篇文章“Predicting Positive and Negative Links in Online Social Networks”说到过一种基于用户反馈的推荐“你可能认识的人”的推荐算法，他把邻近三人之间的三角关系总共16种正负反馈的组合当作特征向量用来表 达用户A和被推荐目标用户C之间的正负反馈，在图里去掉一些已知正负反馈的边来构建训练数据，用简单的Logistic Regression训练模型达到了不错的效果。可以谷歌找到这篇文章的幻灯片，里面有图示讲解。

结语

这篇文章就简单提及一些特征工程的常用方法，说的是手工提取特征，从这些入手可以深入研究研究具体问题的具体做法，这是一个很细致的工作可以多钻研 钻研。这里面没有说到自动提取方法比如深度学习和卷积网络等等，也没说到推荐系统的其他方面比如大规模用户聚类。构建一个推荐系统需要涉及很多东西，绝对 不是GraphLab或者Mahout跑个协同过滤就能上马的，这里面特征工程是很重要的一部分工作，在很多其他数据和机器学习的工作里面特征工程也差不 多是从根本上改变模型效果的重要办法之一。大家在欢乐调参的过程中不妨考虑考虑搞搞特征。

原文链接：http://phunters.lofter.com/post/86d56_194e956

闲聊Openstack贡献

作者：陈沙克

上个月参加Openstack的北京聚会，当时谈到Openstack的贡献，会场上有点争议，到底是Redhat第一，还是HP更多，由于Scope不同，所以得出的结论是不一样的。

Openstack所有的资料都是公开的，包括他们开会，代码提交，审核。那么如何评估那个公司贡献最大，那么这个应该是玩大数据的擅长的东西。我本人压根不懂什么大数据，不过如果你不熟悉开发项目的流程，不熟悉Openstack，仅仅是靠Hadoop，还是没戏的。

Openstack所有的资料都是公开，获取数据不是问题，如何去分析，那就是一大难题。不同的角度，不同的时期都会不一样，代码的贡献，影响力的贡献都是不同的，如何算出一个最大，估计没人可以回答这个问题。这就是说，大数据不是什么万能的药。

对Openstack的贡献，也不需要紧紧理解成代码的贡献，你推广的贡献，文档的贡献，都是必不可少的。中国对Openstack的贡献也不少， 代码上不多，不过在Openstack使用上是很多的。你无法相信，Openstack官方网站，访问最多的是来自中国，有几个英文的网站会是这样的呢?

还有一点就是在Open-
stack讨论贡献最大，是一个比较残酷的事情。因为Openstack 1年2个版本，一直在不断更新，如果你不跟进，很快就会没有人能记住你。这样的故事已经不少了。当年Nebula拿到2500万美金，风光一时，现在基本已经没啥消息。

影响力贡献

现在大家都在谈论Openstack，关注Openstack。那么哪家公司对这个贡献最大呢?

1. HP推出自己的HPcloud，让大家可以直接体验Openstack
2. Rackspace把自己的云平台迁移到Openstack上，让大家对Openstack有信心
3. Mirantis把Openstack包装成产品，服务客户，成为Openstack最大集成商。
4. Redhat派出大量的工程师，给社区贡献代码，完善自己的产品
5. IBM，抛弃自己的smartcloud，整个公司的方向都转移到Openstack上。

如果你说上面那家公司对Openstack影响最大呢？我个人观点，其实还是IBM。他体型庞大，对业界的影响是难以估量的。Openstack的基金会可以成功成立，IBM在作用的非常大的。

Redhat加入基金会,我个人相信都是受IBM影响。

目前每个月一次北京这边的Openstack聚会，都是IBM赞助，也是提高自己的影响力。

代码贡献

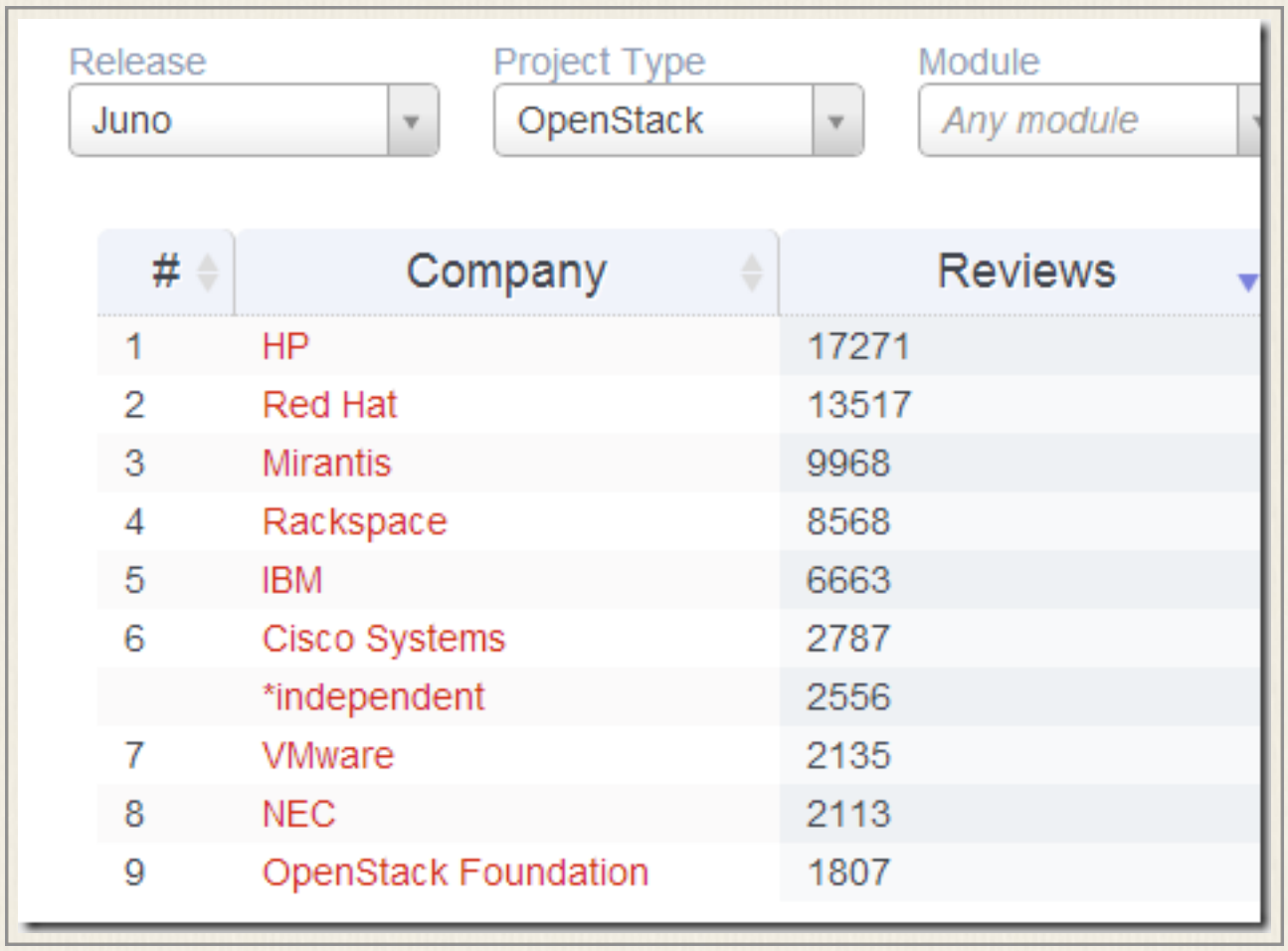
谈到代码贡献，那么你就肯定要看看http://stackalytics.com/ ，这是Mirantis维护的网站，也是Openstack的代码贡献的一个风向标。一个公司投入多少，基本可以通过这里获得。不过由于不同的维度，结论是不一样的，那个维度是最重要的，很多时候，需要你自己去判断。

Stackalytics的统计变化的过程是非常有意思的。这个统计是从Openstack基金会成立后搞的，印象中应该是G版本有的，大家都是通过这个，了解别家干了多少，我做了多少。

下面的图是我2014年8月11日截的，我个人感觉应该还是非常能代码当前Openstack代码贡献的排行榜。

HP和Redhat是一个级别

Mirantis，Rackspace，IBM是一个级别。后面的差距都已经有点大。



#	Company	Reviews
1	HP	17271
2	Red Hat	13517
3	Mirantis	9968
4	Rackspace	8568
5	IBM	6663
6	Cisco Systems	2787
	*independent	2556
7	VMware	2135
8	NEC	2113
9	OpenStack Foundation	1807

代码行数

这是最开始的时候，评估一个项目的贡献指标。这个也比较直观。在Grizzly版本的时候，Openstack功能还是缺少很多的年代。当时Redhat刚刚加入，所以这个版本代码量衡量贡献，其实是没啥问题的。

不过很快就有人发现问题，移动一个目录，相当于删除和增加一堆代码，所以大家当时认为Redhat在灌水。不过我同事还是肯定的态度，经过Redhat的调整，目录是比以前清晰很多，效果还是很好的。

commit次数

到了H版本，Openstack有大量的bug需要修复，如果你修改一个bug，可能就是几行代码。后来经过研究，默认的贡献是用commit次数来衡量。

这个时候，大家都比较关注排行榜，有的公司甚至通过这个榜单来做员工的质效考核。也就导致不少公司在刷榜。一个经典的案例就是：把openstack改成OpenStack，也是一个commit。不是不可以修改，而是不应该为了数量而去做这些修改，这些要让新手，第一次提交的人来 做。

如果commit是为了获取一张门票，那其实还是可以理解，如果是为了刷榜，意义不大，群众的眼睛是雪亮，项目的PTL负责人其实心里都明白，知道那个是刷榜，那个是能干活的。

在Icehouse版本里，还是用commit次数作为指标，这个时候，已经显得不太符合Openstack现状。

Reviews

Openstack的开发者都有点痛苦，提交的代码，很难找到人reviews，来回的修改，reviews，时间太漫长。一段代码需要2位 core review，同意，才能merge。很多人因为坚持不下去，放弃。这个阶段是写代码的人太多，Review代码的人太少。

Mirantis肯定也注意到这个问题，在Juno版本里，把网站的贡献统计改为Reviews次数。鼓励大家去reviews代码。可以发现目前Juno版本的Review大大改善，BP merge的速度也大大提升。

比较一下，这是Icehouse发布后的review数量。

Show 10 entries Search:

#	Module	Reviews
1	nova	18285
2	neutron	8326
3	tempest	6502
4	heat	6496
5	config	5505
6	horizon	4968
7	openstack-manuals	4882
8	keystone	4464
9	cinder	4078
10	ironic	3430

这是目前Juno版本，还有2个月的时间，才正式发布。

#	Module	Reviews
1	nova	9640
2	neutron	7553
3	config	5407
4	horizon	4149
5	heat	3794
6	tempest	3532
7	keystone	2656
8	ironic	2581
9	cinder	2407
10	tripleo-image-elements	2185

Juno正式发布的时候，review数量，应该会比Icehouse版本高出很多。大家可以重点关于Nova，Neutron，和 Horizon。Juno版本的Neutron和Horizon的review数量，已经基本达到Icehose发布时候的数量级别。说明这两个项目的 PTL很给力。Neutron的PTL刚换成思科的。

通过Reviews数量，其实你可以了解到更多信息

1. 项目的活跃程度
2. 项目的core的数量
3. 项目的成熟程度

当然，Reviews数量，肯定也有人想到钻空子的办法，看到PTL+2，赶紧上去给一个+1，这个时候是没任何风险的，而且代码都不用看，哪家公司的就不说了，所以有时候大公司的，真的要管好，不然也挺丢人的。

总结

目前Openstack的所谓核心项目大概有10个左右，每个项目都有一个负责技术的PTL，大概有10个左右的core。Neutron和 nova，core数量相对多

一点，达到15个左右。Openstack真正Core，在干活的，应该就100位左右，目前中国区应该在5到8个左右。

如果从代码贡献来说，尤其是Redhat把eNovance收购以后，我个人的观点目前排名应该是

1. Redhat
2. HP
3. Mirantis
4. IBM
5. Rackspace

目前HP投入很大，基本能和Redhat 一个档次上。不过能持续多久，这个就不好说。

国内的代码贡献

如果说Openstack是下一个linux，那么国内目前在代码贡献上，其实不算太落后。还没有产生项目的PTL，不过Core的数量，其实已经不算太少了。

国内外企

目前国内IBM，Intel，Redhat，Ubuntu都是有中国区的员工在提交代码，Intel还有4位的Core（没记错，cinder 1位，Ceilometer 2位，Olso 1位）IBM中国区Core基本都已经流失了不少，认识的2个中国区core，都已经离开IBM

国内企业

目前国内参与Openstack社区开发的不少。要多说的是以前华为一直都是给自己产品增加驱动，对Openstack贡献不大在Icehouse 版本里没有完成一个BP。在Juno版本里，目前已经完成了18个BP，在这真的是一个非常大的进步。在Juno版本完成18个BP和以前的版本完成是不 一样的概念，现在Openstack的组件的功能，已经相对完善，要去完成一个BP的难度其实很

大，我相信下面很多BP，不少应该都是已经搞了1年。华为 在没有自己项目的Core的支持下，完成那么多BP，是非常难得的。

#	Module	Completed Blueprints
1	heat	5
2	nova	4
3	cinder	3
4	python-heatclient	2
5	tempest	1
6	python-cinderclient	1
7	python-novaclient	1
8	ceilometer	1

上图是华为完成的BP，和以前已经很大不一样了，例如Cinder完成的BP，并不是加自己的存储的驱动，而且真的在做东西，给社区做东西。都是很有价值的东西。

下面就是华为完成的Cinder的BP介绍，至少我感觉是很有价值。

ling-yun (Huawei)
14 Jul 2014 00:24:34 in cinder
Blueprint "Support Volume Num Weighter" (support-volume-num-weighter)
Currently cinder support choosing volume backend according to free_capacity and allocated_capacity.
Volume Num Weighter is that scheduler could choose volume backend based on volume number in volume backend, which could provide another mean to help improve volume-backends' IO balance and volumes' IO performance.
Explain the benifit from volume number weighter by this use case.
Assume we have volume-backend-A with 300G and volume-backend-B with 100G.
Volume-backend-A's IO capabilities is the same volume-backend-B IO capabilities.
Each volume's IO usage are almost the same.
Use CapacityWeigher as weighter class.
Concrete Use Case
If we create six 10G volumes, these volumes would placed in volume-backend A. All the six volume IO stream has been push on volume-backend-A, which would cause volume-backend-A does much IO scheduling work. At the same time, volume-backend-B has no volume and its io capabilites has been wasted.
If we have volume number weighter, scheduler could do proper initial placment for these volumes----three on volume-backend A, three on volume-backend-B. So that we can make full use of all volume-backends' IO capabilities to hlep improve volume-backends' IO balance and volumes' IO performance.
Priority: Medium
Status: Complete (Approved, Implemented)
Mention count: 1, last mention on 03 Jul 2014 17:49:15

Rui Chen (Huawei)
30 May 2014 02:46:18 in cinder
Blueprint "lock volume" (lock-volume)
We need a feature to lock volume, aims to avoiding to delete volume accidentally.
Priority: Undefined
Status: Complete (Obsolete, Unknown)

wingwj (Huawei)
25 Apr 2014 07:31:23 in cinder
Blueprint "Support Glance v2 API" (use-glance-v2-api)
Now cinder only supports glanceclient v1 by default. The version should be able to configurable via a configuration parameter to use either Glance V1 or V2 API.
Due to differences in the workflow exposed by the API, the image service layer will need to be able to present a consistent interface to Cinder while performing necessary translations to interact with the configured Glance API version.
Priority: Undefined
Status: Complete (Obsolete, Unknown)

1. 华为
2. unitedstack
3. easystack
4. 麒麟
5. 网易

对Openstack项目有真正有点影响力的是Horizon，Ceilometer，Heat。至少项目的负责人记住你，你真的是对项目做过贡献。对于开源社区来讲，都是个人，不看你公司。

整体来说，我想中国区对Openstack的代码贡献，应该在5%左右，华为应该3%，超过一大半以上。这也是完全是自己个人感觉的结论。

原文链接：http://www.chenshake.com/chat-openstack-contribution/

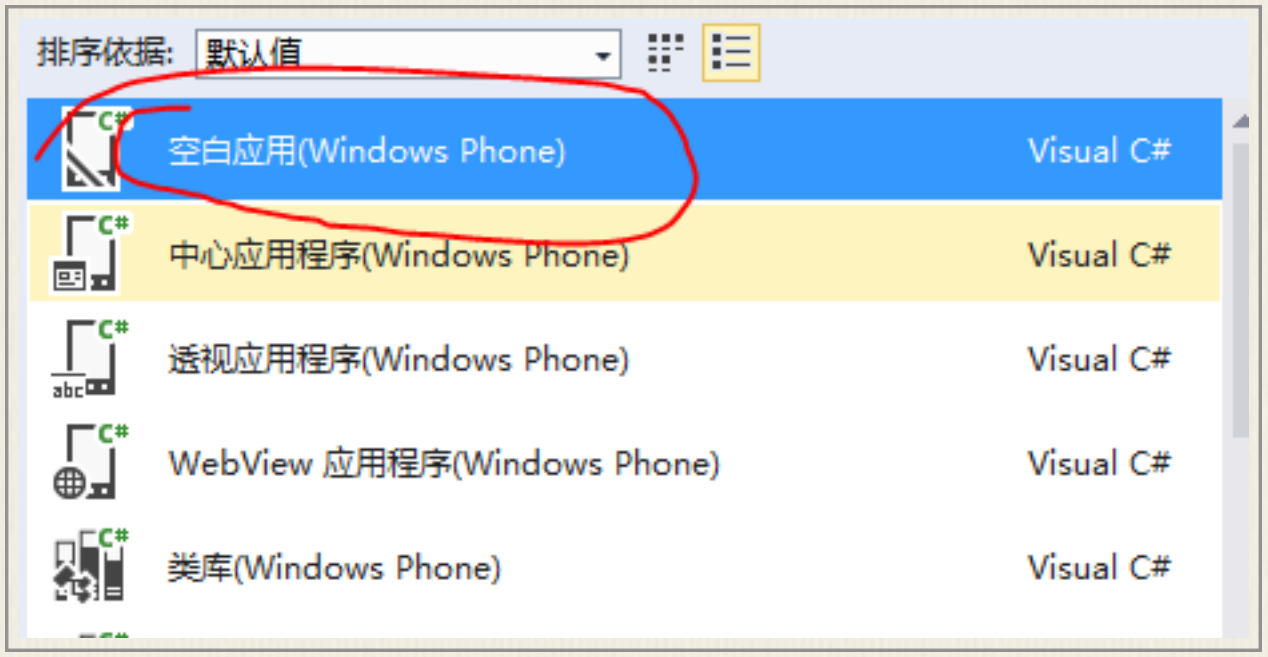
【WP 8.1开发】手机客户端应用接收推送通知

作者：东邪独孤

上一篇文章中，已经完成了用于发送通知的服务器端，接下来我们就用这个服务端来测试一下。

在开始测试之前，我们要做一个接收通知的WP应用。

1、启动VS Express for Windows，新建项目，在项目模板中选择“空白应用程序（Windows Phone）”。



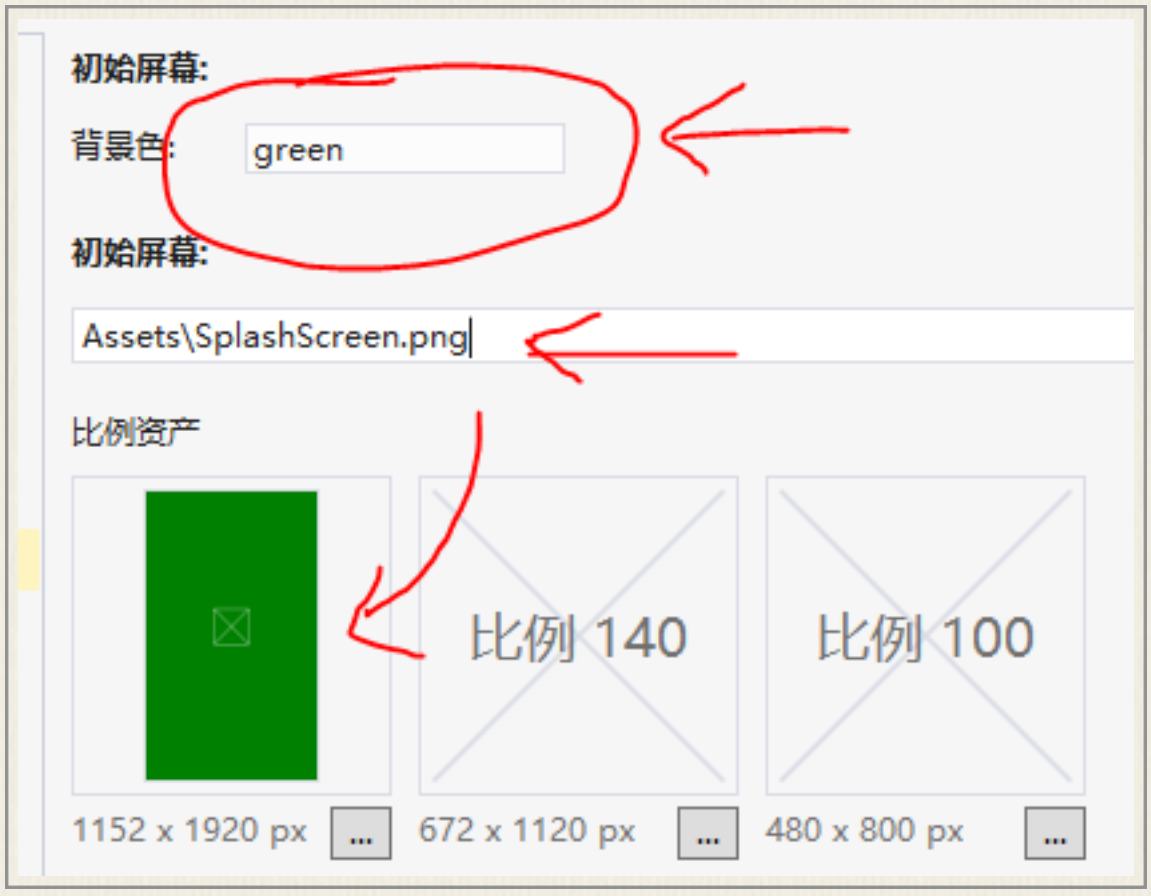
2、既然要接收通知，肯定少不了Toast、磁贴这几样常用的通知的，故我们得先准备一些图片。

在“解决方案资源管理器”中，双击打开清单文件，切换到“可见资产”选项卡，这个“资产”指的不你的银行卡存款有多少，而是你的应用中的一些如图片、音乐等资源，可以不太好翻译，就按单词直译了，反正你知道它是啥就行了。

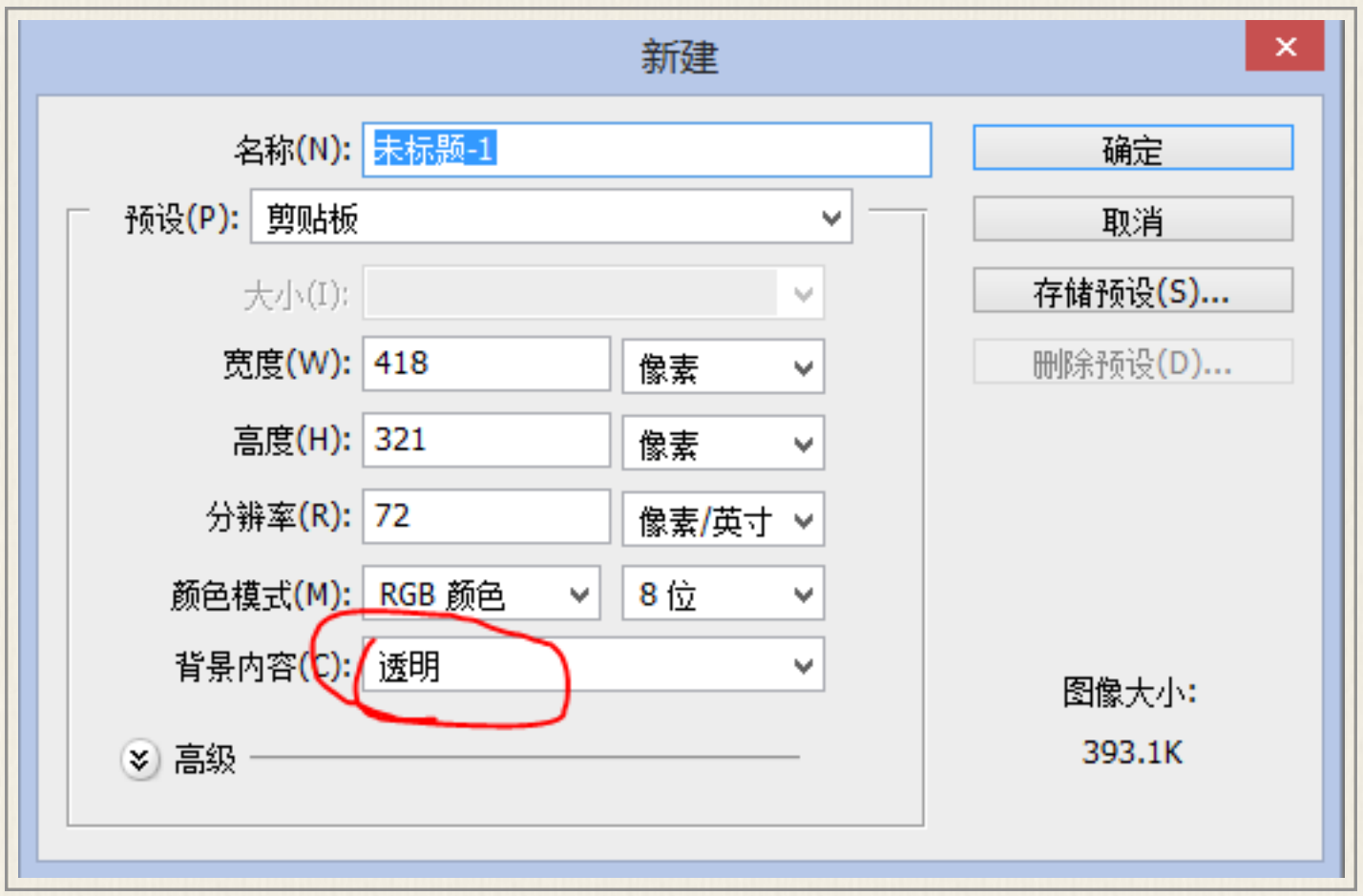
我们看到，种图标所需的尺寸都已经注明，注意每一类图标都有三种比例，分别为100%，140%和240%。所以，对于每一种图标，理论上我们需要分别准备三张图片，假设图片名字为abc，用于100%比例的图片可以命名为abc.scale-100.png；对于140%比例的图片，可以命名为abc.scale-140.png……中间多了个scale-XXX，XXX就是比例值。

如果你觉得麻烦，可以只为一种比例准备图片即可，比如我只准备100%的所有图片，不过，我们都知道，图片放大后会变模糊，但缩小后不会变模糊。所以，我们应用选用比例最大的（240%）的标准来准备图片文件。

举个例子，初始屏幕，从窗口中的提示我们看到，240%比例所需的尺寸为1152×1920，我们就设计一张这样大小的图片，命名为xxx.scale-240.png，最好用PNG图片，因为它允许背景透明，通常我们应当考虑使用透明背景，初屏幕的背景颜色可以另外设置。如下图所 示，为了支持环保事业，我把初始屏幕的背景色改为绿色。

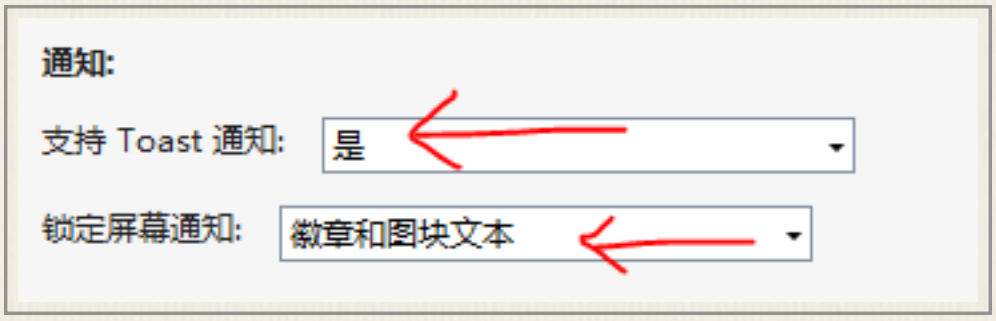


这个例子用来测试，也不用设计得太认真，打开PS，随便涂鸦几张图片就行了。为啥要用PS？有人说，用系统自带的画图不就行了吗？但是，你要知道，系统自带的 画图 程序在绘制PNG图像时，背景是非透明的，但PS在创建新内容时，可以选择透明背景，如下图所示。

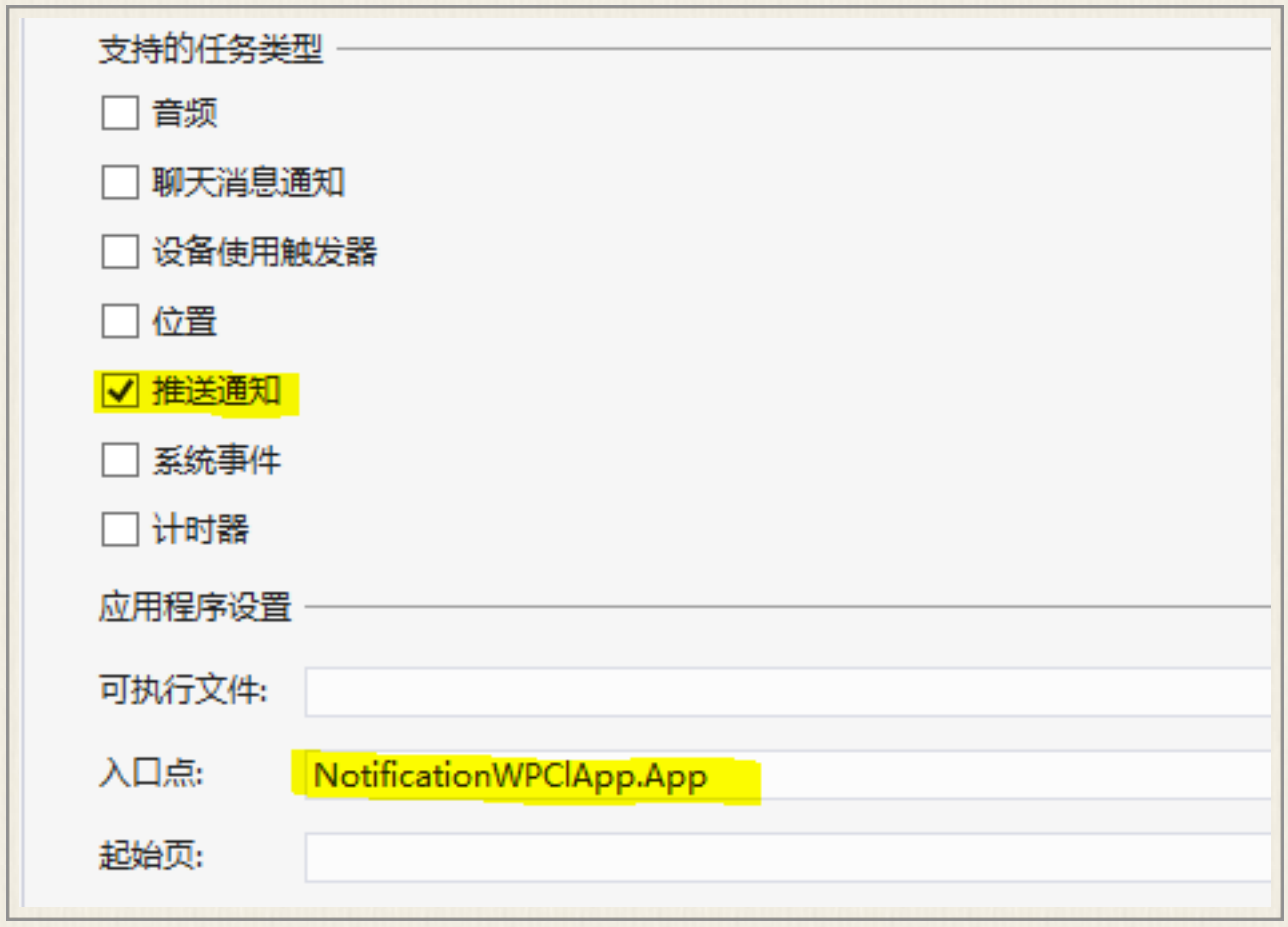


有人又会问，PS是什么？PS就是PhotoShop的简称。

3、咱们干脆把锁屏也用上，切换到“应用程序”选项卡，然后找到“通知”节，开启Toast通知，并把锁屏通知改为徽章和图块文本。



4、开启锁屏提醒后，必须设置后台任务，这个我们以后再扯，本示我们不需要后台任务，但也不能空在那里，切换到“声明”选项卡，增加一个后台任务，在支持 的类型处勾上“推送通知”；由于我们没有开发后台任务，因此在入口点处填上当前应用中App类的名字，包括命名空间名称。



保存并关闭清单文件。

5、打开MainPage页面的代码视图，在OnNavigated-To方法中加入以下代码。

```
protected async override void OnNavigatedTo(NavigationEven-tArgs e)
{
    PushNotificationChannel channel = await PushNotificationChannelManager.CreatePushNotificationChannelForAppli cationAsync();
    // 如果本地设置中没有相关键，表明是第一次使用
    // 需要存储URL，并发送给服务器
    //if (Windows.Storage.ApplicationData.Current.LocalSettings.Values.Contains Key("url")==false)
    //{
    // Windows.Storage.ApplicationData.Current.LocalSettings.Values["url"] = channel.Uri;
    //    SendURL(channel.Uri);
    //}
    //else
    //{
    //    string savedUrl = Windows.Storage.ApplicationData.Current.LocalSettings.Values["url"] as string;
    //    // 当URL改变了，就重新发给服务器
    //    if (savedUrl != channel.Uri)
```

```
            //    {
            //        // 再次保存本地设置
            //
Windows.Storage.ApplicationData.Current.LocalSettings.Values["url"] =
channel.Uri;
            //        SendURL(channel.Uri);
            //    }
            //}

            System.Diagnostics.Debug.WriteLine(channel.Uri);
            SendURL(channel.Uri);
        }
```

调用CreatePushNotificationChannelForApplicationAsync方法创建推送通道，然后把通道的URL发送给我们自己写的服务器，服务器就是根据这个来向手机发送推送的。

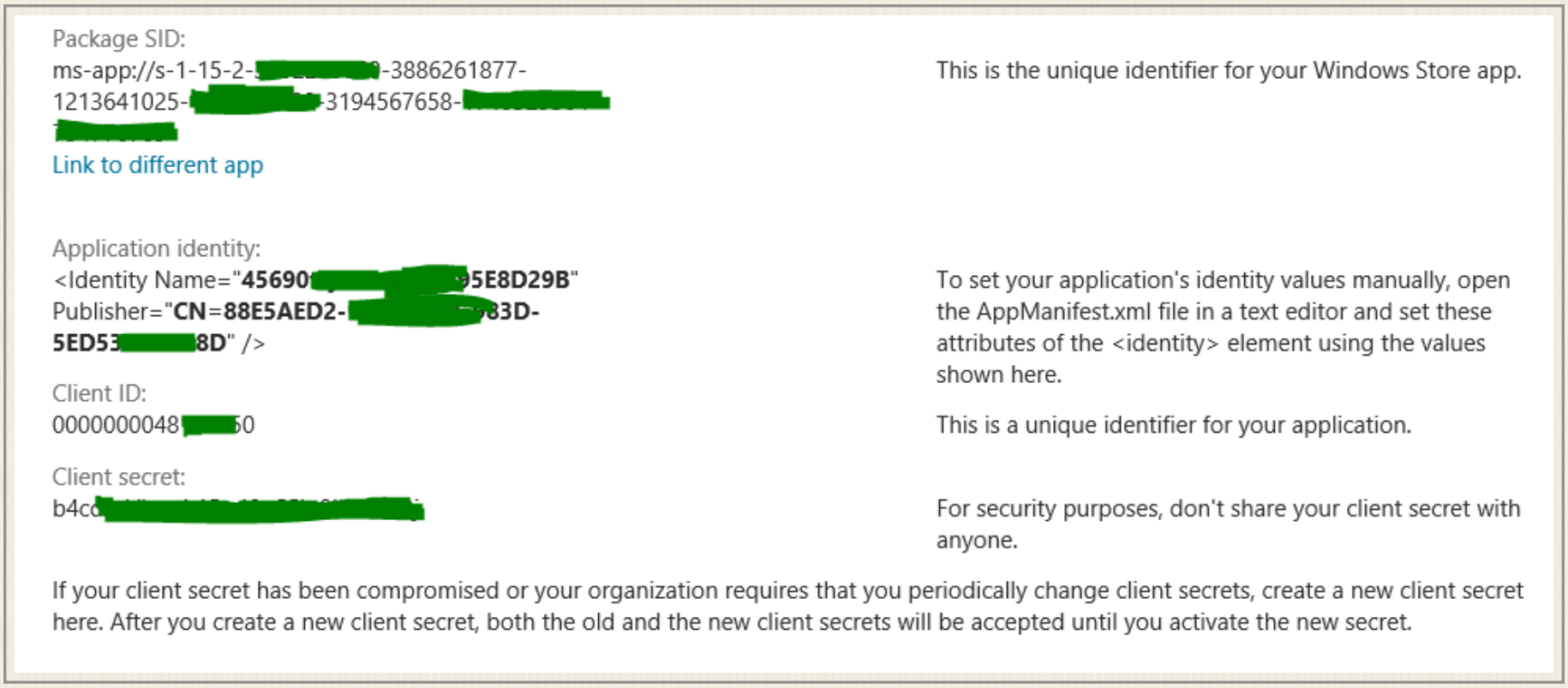
注意被注释掉的那段代码，作用是把获取到的通道URL保存到本地设置中，如果获取到的新URL和本地设置中的URL相同，说明URL没有改变，就不必把URL发给服务器了。

SendURL方法的作用是把通道URL发送给服务器，代码如下：

```
 private async void SendURL(string url)
        {
            using (HttpClient client =new HttpClient())
            {
                byte[] data = System.Text.Encoding.UTF8.GetBytes(url);
                ByteArrayContent content = new ByteArrayContent(data);
```

```
                try
                {
                    await client.PostAsync("http://192.168.1.100:85/svr/", con-
tent);
                }
                catch { }
            }
        }
```

好了，我们就剩下最重要的一步，就是设置应用程序的清单文件。还记得上一文章中，我们创建应用时得到的SID，App ID等几个ID吗?

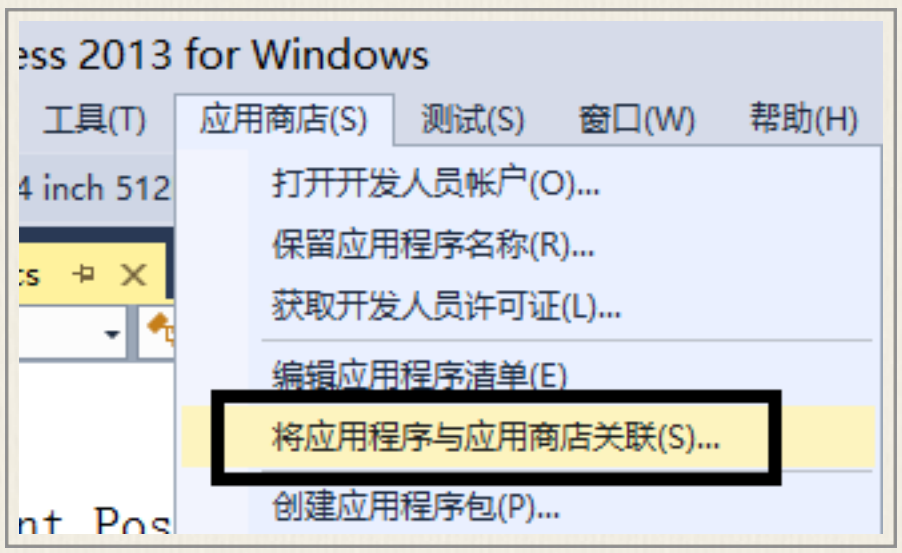


在服务器端，使用这些信息来申请access token，在WP应用中我们也同样需要把这些信息写到清单文件中，这样推送服务器才能进行推送，如果标识不匹配，就不会进行推送，防止有些别有用心的人自作多情，到处推送垃圾情书，造成信息污染。

那我们是不是打开清单文件，然后一个个改吗？你要是原意的话，也无所谓。但是，我们是21世纪的高大上，不需要机械劳动，下面我给大家演示一下，如 何智能地把商店中的应用信息同步到清单文件（如果你的应用已上传到应用商店，就不需要这样做了，但是在测试或学习阶段，不要上传）。

在“解决方案资源管理器”中右击项目名，从快捷菜单中选择“应用商店”->“将应用程序与应用商店关联”，如下图

或者，在VS的菜单栏中依次执行“应用商店”>“将应用程序与应用商店关联”菜单。

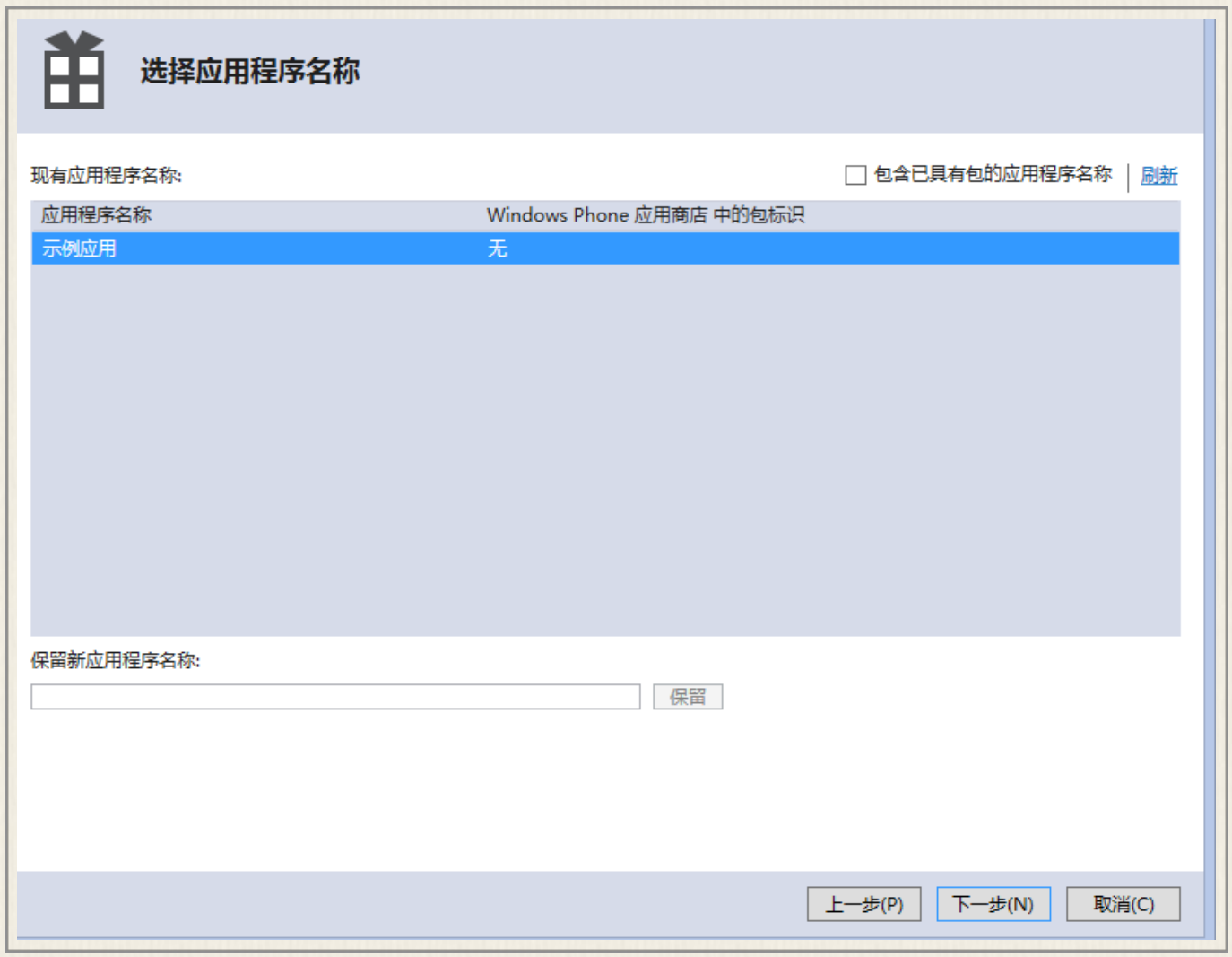


这时候会出现一个向导，点 下一步。

这时需要输入开发者帐号进行验证。

验证成功后，正在连接应用商店。

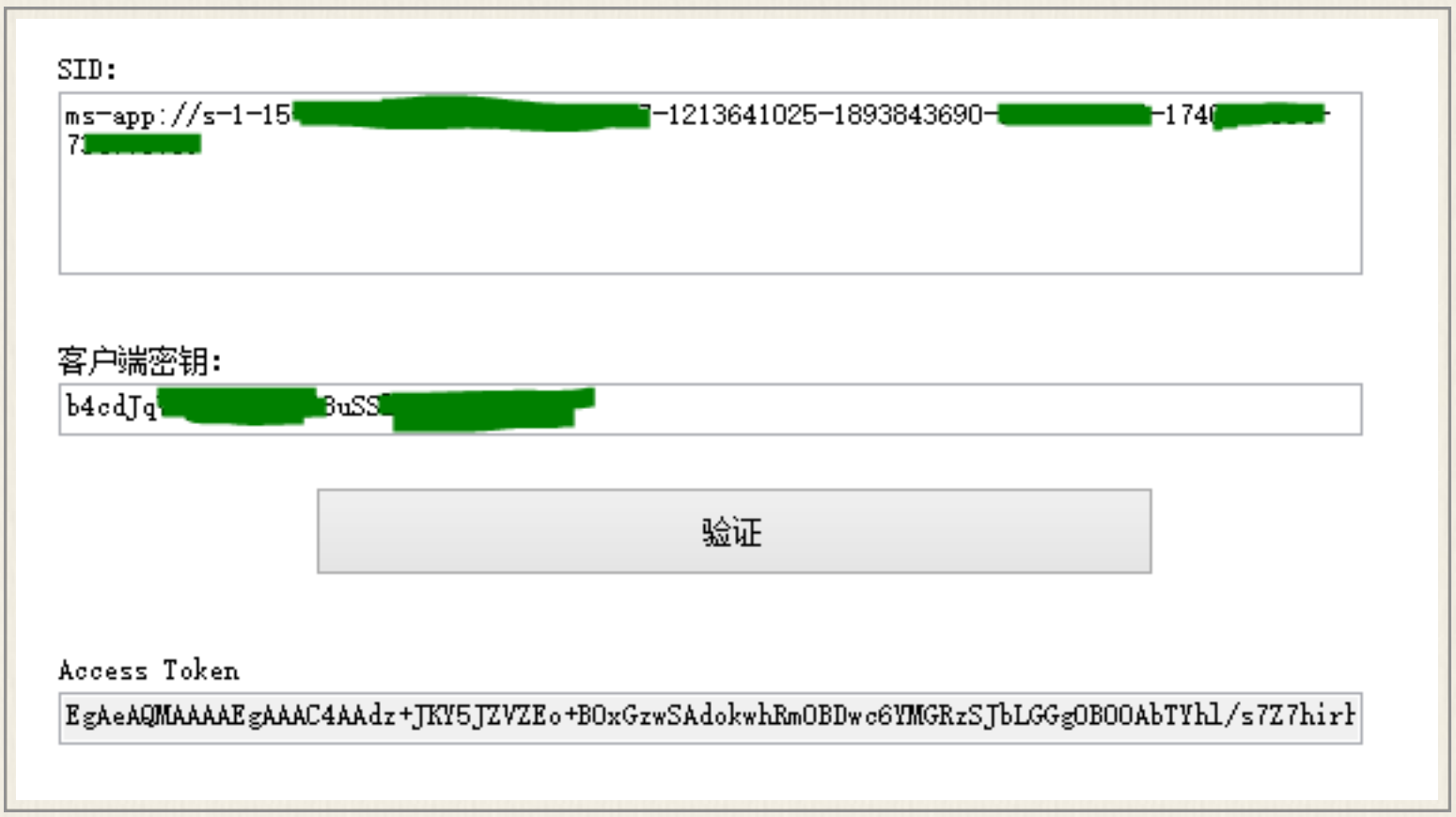
然后，在应用列表中选择你需要同步的应用，这里我还是选上次创建的“示例应用”。



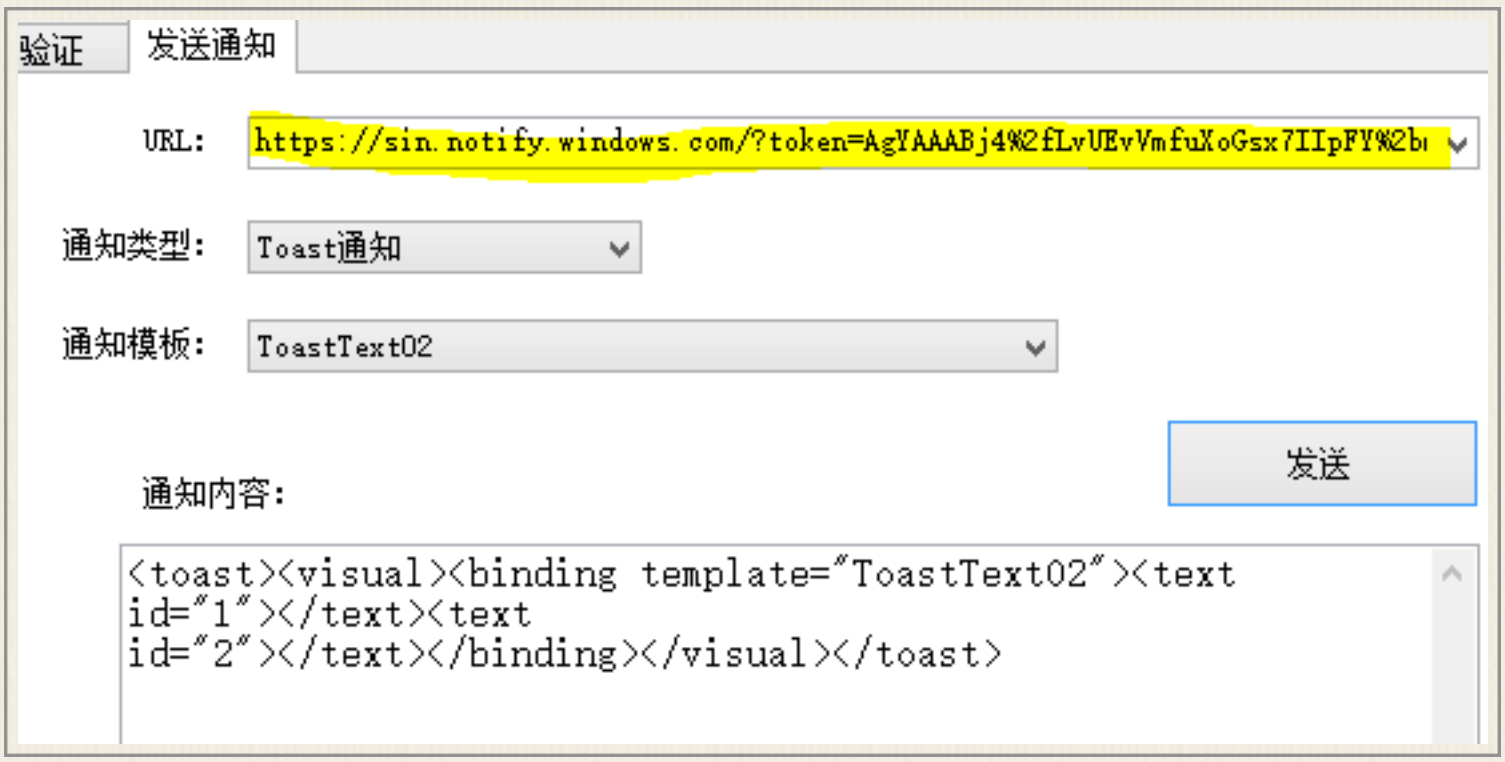
然后一路 下一步 ， 直到完成为止。应用程序清单文件会自动修改。

==========

下面我们就可以测试了，首先记得要以管理员身份运行我们前面写的服务器。输入SID和客户端密钥，获取access token。



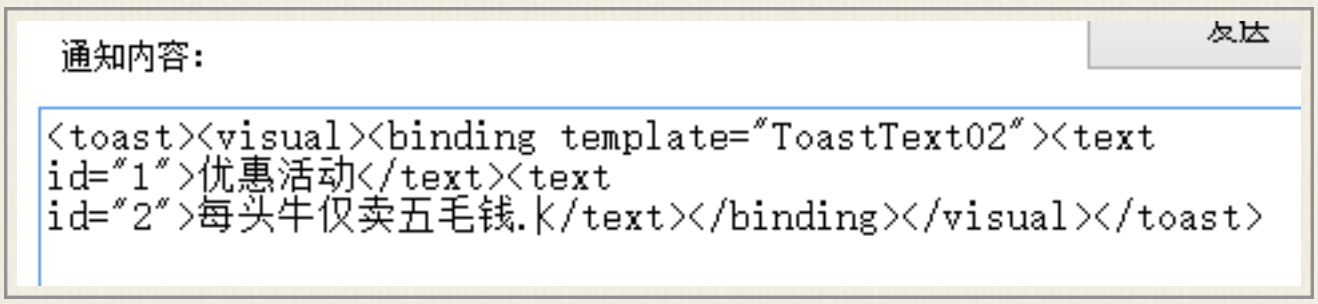
运行WP手机客户端应用，会获取到通道URL并发送给服务器。



测试Toast通知

对WP来说，Toast只有ToastText02这个模板可用，就算你使用其他模板，它依然强制使用ToastText02模板。

修改XML模板，id为1的text无素设置toast标题，标题将以粗体显示；id为2的text元素为内容，显示为正常字体。

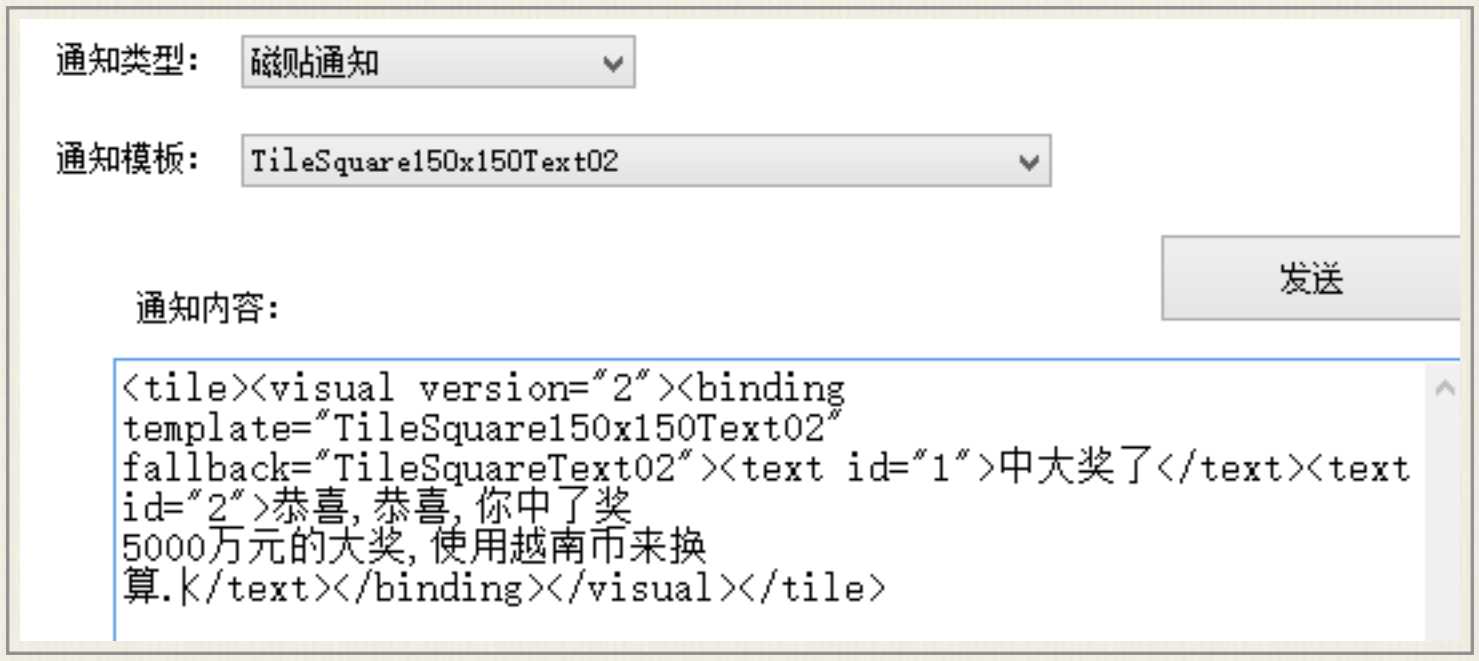


点击发送后，在手机上你会看到奇迹的发生。

如果使用X-WNS-SuppressPopup 标头并设为true，Toast通知不会弹出，而是直接扔进操作中心队列中了。

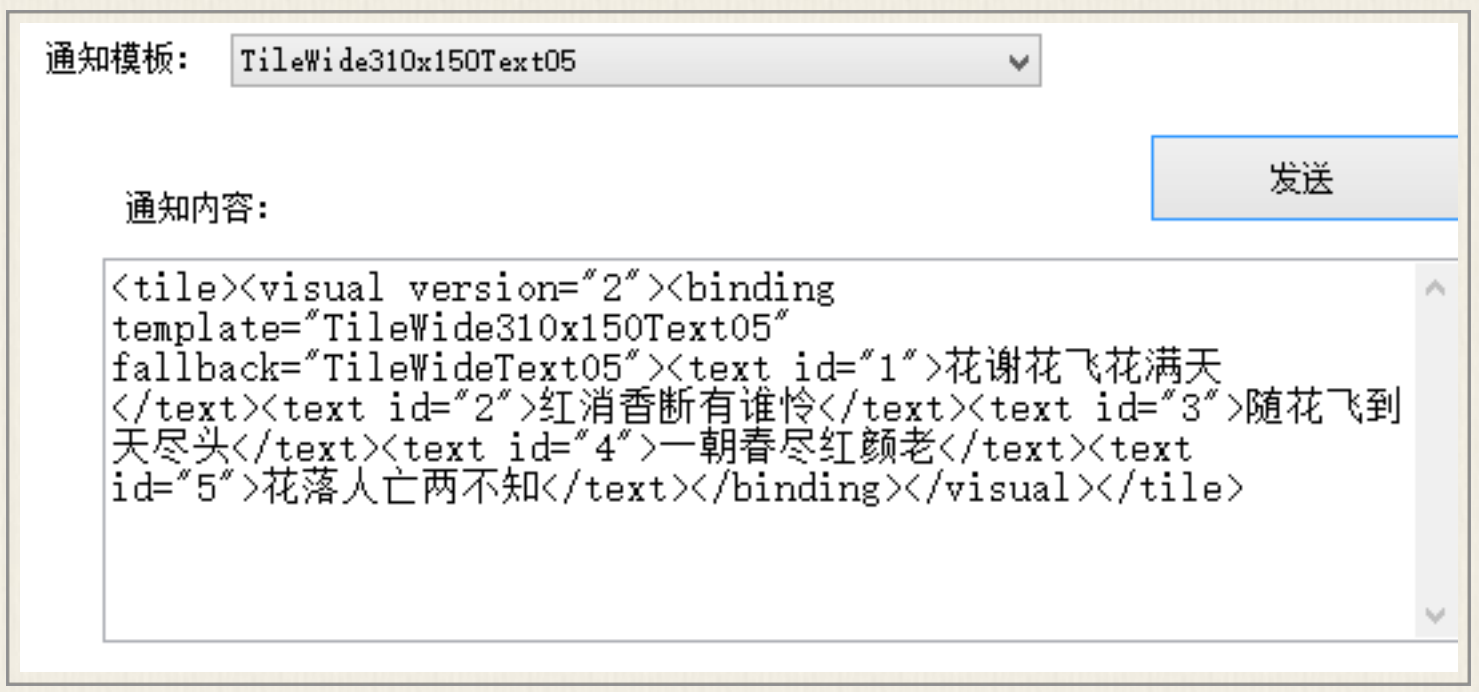
测试磁贴

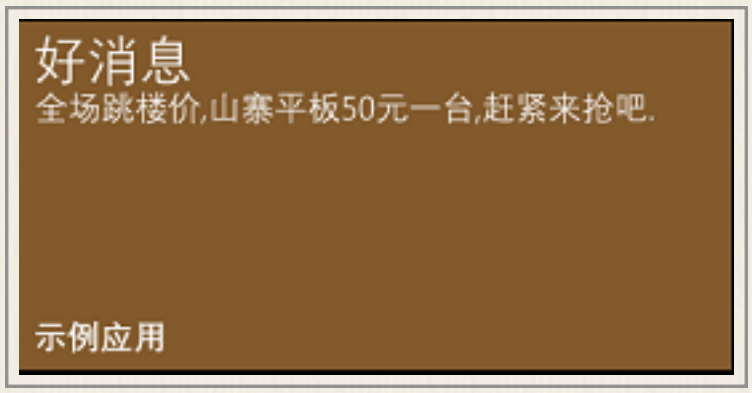
要看到磁贴通知，先要把应用固定到“开始”屏幕。

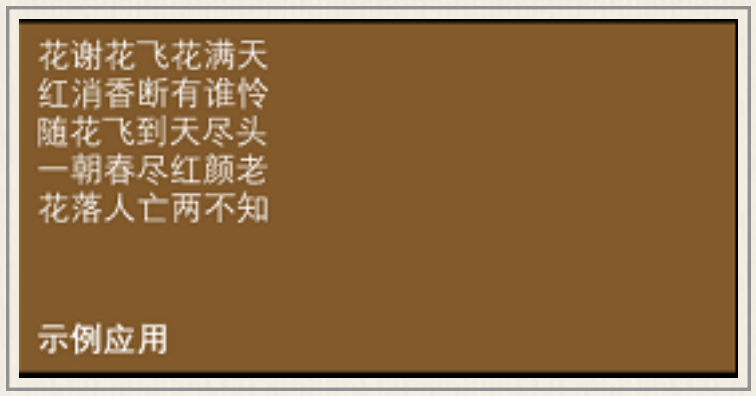


这时候，会看到手机“开始”屏幕上的磁贴已经更新。

也可以试试宽磁贴。

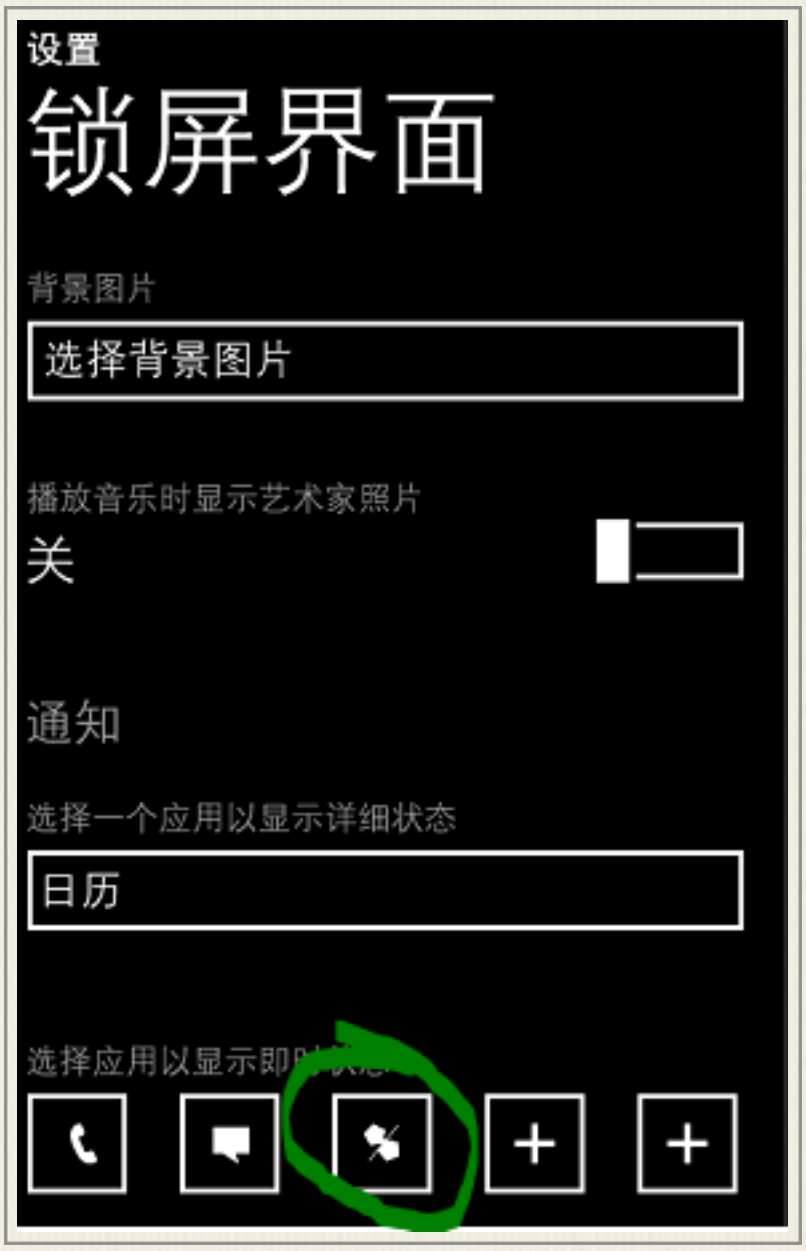




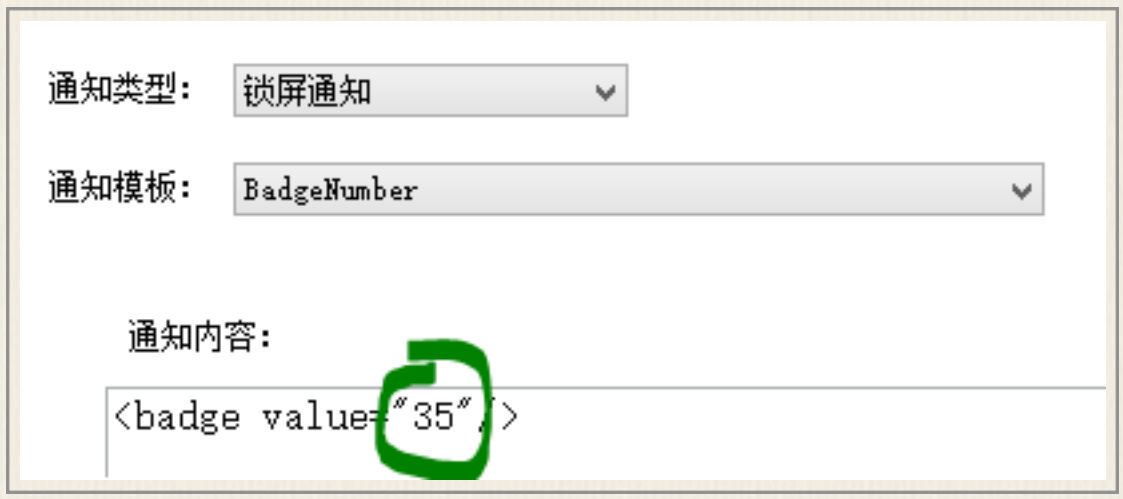


测试锁屏通知

先到系统设置中，把应用程序加入到锁屏序列中。



然后把手机锁屏，就可以测试了。



请看屏幕下方。

源码下载：http://files.cnblogs.com/tcjiaan/NotificationWPClApp.rar

对于RAW通知，可以与后台任务结合起来，即通过后台任务在后台来接收。关于这个，我们下一篇文章再扯。

原文链接：http://www.cnblogs.com/tcjiaan/p/3905090.html

从Bitly构建分布式系统中吸取的教训

译者：潘瑾瑜

在五月的Bacon会议中，bitly应用开发主管Sean O'Connor 讲解了bitly开发人员从构建一个月处理点击量60亿的分布式系统中吸取到的相关教训。

什么是分布式系统?

Sean表示，关于分布式系统的定义很容易在维基百科上找到，一个分布式系统有三个很明显的特征：

- 组件节点间真正的并发，要有相关的开销和复杂度来协调这些节点。
- 缺少一个共有的时钟，让不同节点上的事件按时间顺序发生是不可能的。
- 独立的故障，应该被理解为具有这样的能力：一个节点的失败不会影响到系统中的其他节点。

构建一个分布式系统需要处理好这些特性。Sean认为这样的一种方法已经过时了。这种方法通过将系统从一开始就抽象成不是分布式的系统，隐藏了一个 系统的分布式特性带来的复杂度。Sean说，由于系统的分布式特性迟早会从抽象中显现出来，这种方法注定会失败。取而代之的，抽象应该建立系统分布式特性 的模型，并处理好任何显现出来的特性。

将服务作为积木

Bitly的架构使用服务的概念进行定义。一个服务是通过API明确定义的、专门提供一些功能的抽象。根据Sean的描述，服务对应于分布式系统就像函数对应于代码，这就允许站在更高的层次上看问题，而不用知道所有的实现细节。这个想法带来了另一种思考问题的方式。

Sean提到了通过小型服务构建系统所获得的益处：

- 代码行数的减少，使得一个服务更容易被理解。
- 独立的故障，因为如果一个服务停止只意味着系统失去了特定的功能，但整体上系统仍然保持运行。
- 易于判别系统的哪部分存在问题。

异步消息传递

确保一个系统可扩展的关键点是，在节点之间使用异步消息和消息队列来改进节点的独立性，所以一个节点没有必要等待另一个节点的回应。

这样做的一个好处是，如果一个节点出现了一些问题而暂时不能处理所有到来的消息时，这些消息将被保存在队列中以便尽快处理。此外，一个节点的故障不会直接影响其他节点。

尽管在许多场景中，异步消息传递更自然地用于同步处理某一类操作，但它也有复杂之处。比如Sean提到，在bitly，由于对数据处理有越快越好和 一致性的要求，缩短URL是一个完全同步的操作。这意味着一个缩短后的URL不应该返回给不同的使用者。另一方面，分析有着不同的需求，适合于使用完全异 步的操作执行。因此，当bitly想要收集处理关于用户在链接上的行为的度量数据时，它只是将数据放进流向下游处理的队列中，而不用关心处理需要花多长时 间。根据Sean的解释，尝试将一个本质上同步的操作异步化可能非常复杂，因此最好能先了解这个操作的本质。

关于节点间的交互，Sean最终的评论指出，相比于将消息看成是从一个节点流到另一个节点的命令，把消息当成事件将更有益。一个事件是对某地发生某 事的一个描述，并且发送节点不需要知道接收节点的任何信息。另一方面，如果一条消息是一个命令，那么发送节点必须了解接收节点处理这条命令的能力。因此， 在生产者节点不知道任何信息的情况下，把消息看待成事件极大地有助于节点隔离、天然地支持多消费者以及消费者的动态增加和删除。

把消息当成事件也会导致另外一种实现：从生产者的角度看，标记消息的方法比过滤消息的方法更好。作为一个例子，Sean提到了私有和公有链接的处 理。生产者可以对私有链接进行过滤，以确保这些链接不会到达对

其不感兴趣的下游节点。但这要求生产者能推断出下游需要关心什么样的数据。取而代之 地，bitly对私有链接进行标记，让消息自由地流入下游，确信下游节点能够以适当的方式对消息进行处理。

让服务能良好地运行在一起

Bitly通过以下方法确保服务良好地运行在一起：

- 使用排压机制，让发送请求的节点知道一个服务是否繁忙或过载，这样它们就能够调节发送请求的速率。这有助于维持系统的健康，防止连锁错误的发生。作为一个例子，Sean提到了服务缓存的初始化：如果在缓存初始化过程中不断地接收到请求，就会有导致数据出错的风险。
- 根据服务的健康状态对请求进行负载均衡。Bitly使用了自己开发的hostpool用于负载均衡，跟踪失败的服务并优先选择更健康的服务。

监控

通过引用Leslie Lamport的话“一个分布式系统是一个这样的系统，系统中一台机器上未知的故障都可能会导致你自己的计算机无法使用”，Sean引入了监控这个话题。Bitly拥有400多台服务器，所以监控是一个非常必要的任务，因为这是知道系统中某些东西无法正常工作的唯一途径。

Sean提供了一些关于监控的诀窍：

- 使用Nagios检查服务器状态。
- 运行完整性检查以确保服务提供正确的数据。
- 将日志集中在一起，因为集中不同节点的日志有助于分析和诊断系统中发生了什么。
- 确保相关的人能及时收到相关的信息。Bitly使用了自己开发的nsq消息平台，旨在通过和Web UI的结合，能容易地观察到部署期间的一举一动。

Bitly是一个用于社交网络，短信和电子邮件的缩短URL服务。除了缩短URL，bitly也搜集关于引用链接的分析数据，这是bitly商业模型的核心。

原译文链接：http://www.infoq.com/cn/news/2014/08/bitly-lessons-learned

原文链接：http://www.infoq.com/news/2014/07/bitly-lessons-learned

轻博客始祖Tumblr：哈希以支撑2.3万Blog请求/秒

作者：TodHoff

摘要：Tumblr成立于2007年，是目前全球最大的轻博客网站，拥有超过1.96亿的博客，930亿帖子。同时，该网站Blog请求更已达到2.3万每秒。

【编者按】Tumblr是目前全球最大的轻博客网站，也是轻博客网站的始祖。当下已有超过1.96亿博客，930亿帖子，每秒2万3千请求。近日，该公司网站可靠性工程师Michael Schenck在HighScalablity上公布了其架构设计。

在Tumblr，blog是网站流量最大的一部分。而在tumblelogs中，高度可缓冲成为一个非常重要的特性。鉴于Tumblr支撑的高views/post比率，做到这一点并不容易，下面一起看向blog支撑部分的架构。

状态

- 278个员工，6个人负责Tumblr的perimeter（Perimeter-SRE），包括一个manager。
- 超过2800台服务器，不到20%用于blog支撑
- 峰值期间每秒2.3万blog请求
- 峰值期间每秒6500个blog缓存清理
- 超过1.96亿blog
- 超过930亿post

平台

- HAProxy
- Varnish
- Bird

曾经架构——基于映射的分割

早期，Tumblr运行在一个非常小的规模——1活跃加1备用的proxy 服务器，以及同样配置的varnish节点。这种规模的Tumblr非常易于管理、监控，服务的可用性也非常高。然而不久后，Tumblr就不得不疯狂的扩展以应对用户暴增可能带来的容量限制。

添加1个独立的proxy节点

添加一个独立的proxy服务器非常普遍，同时还涉及了DNS。通常情况下，类似Round-Robin A Record这些基础的东西可能就会满足需求，但是很多情况下，一个健康检查GSLB配置同样值得你投入，即使是在同一个地理位置下。

DNS的缺点是，域名服务器会以一个恒定的速率对每个IP做出响应，然而问题在于并不能保证每个查找都用于相同数量的请求。在1分钟内，用户A对一 个Resolved IP进行的请求可能是10次，而用户B可能会达到100次。如果你有两个IP，A和B分别使用1个，假设只有这两个用户访问，那么一个proxy 服务器的请求速度可能是另外一个的十倍。

这个问题可以通过Lower TTL来解决，比如一个30 second TTL可以将请求在这两个proxy 间平衡。在前30秒，A被送到 P1，B被送到P2，而在后30秒可能就会置换，在最后每个proxy 服务器处理都会处理大约60个请求左右。Lower TTL的缺点在于会造成更多 的查询，因此会带来更多的DNS开销。但是值得庆幸的是，DNS基本上是开销最低的第三方服务。

添加1个独立的 varnish节点

当DNS给你带来更多proxy层上的空间时，varnish的扩展往往会复杂一点。尽管你困扰于并发请求带来的单varnish节点容量限制，但 是简单添加1个varnish节点并不能达到你的预期需求。这个操作可能会降低cache-hit比率，让清理在resource/time上更加密集， 并不能真正的增加你的缓存容量。

迭代单varnish节点最简单的方法就是静态分割，这包括确定你的唯一识别符，并将这些空间在两个节点中分割。对Tumblelogs来说，这就 是blog的主机名称。鉴于DNS区分大小写，你只需考虑40个字符，字母数字“a-z”、“0-9”以及“-”、“_”、“.”、“~”4个字符。因此 对于两个varnish节点，blog主机名称根据首字母在两个缓存节点中分割。

均匀的分布式分割——通过一致性哈希

前面的两个例子（DNS round-robin和静态分割）虽然想法是正确的，但是并未做一个细粒度的分割。在小规模下，这种粒度可能并不会带来 问题。但是随着流量的增长，差异性逐渐的造成问题。减少least-hot 和most-hot节点间差异至关重要，这里就有了一致性哈希的用武之地。

分割proxy流量

如果你的服务器环境满足这个条件，即可以改变路由器（建立于用户和proxy服务器之间）的路由表，那么你可以利用其ECMP的优势。ECMP可以 在一致性哈希环中将proxy分割，然后将请求者们映射到这些分割后的碎片上。通过将多路径（proxy服务器）路由基础设施发送给一个特定的IP（高可 用IP）来实现这个操作，在这里，ECMP将哈希请求源以确定哪个proxy来接收这个请求会话包。典型的ECMP实现提供了Layer 3 （IP-only）和Layer 3+4（IP:port）哈希选项，Layer 3意味着所有特定IP上的请求都会交由一个制定的proxy，这点非常有利于 debug，但是对于使用了单一NAT IP的大型网络来说并不有利。Layer 3+4则提供了非常经典的分布式特性，但是debug指定客户端变得非 常有挑战。

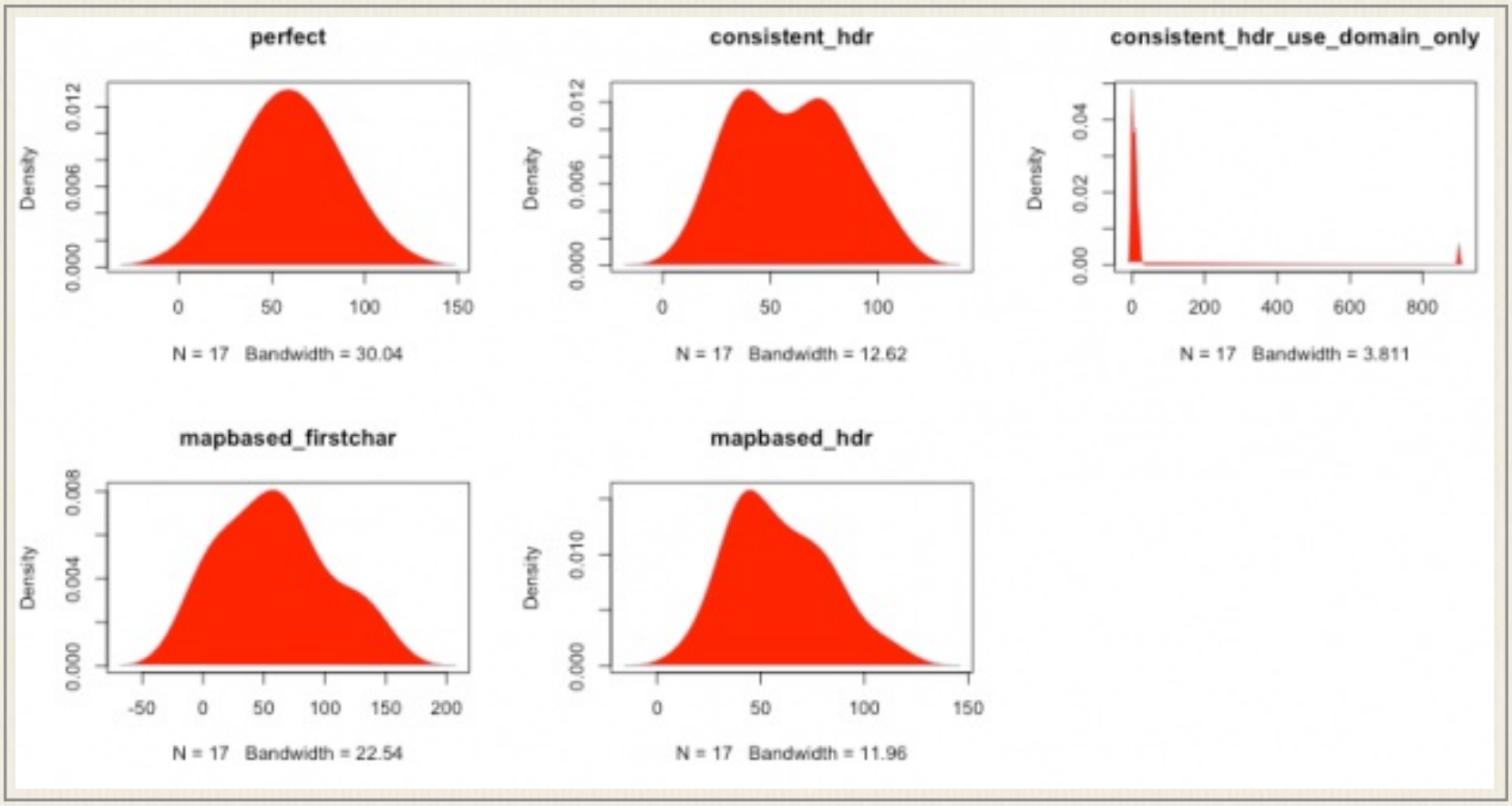
这里有非常多的方法来INFORM多路径路由器，然而我们更推荐使用OSPF或者iBGP来做动态route advertisements。一个 只需要监视loopback interface上的高可用IP，允许内部路由，并将其IP作为一个next-hop advertise给高可用IP。 同时我们还发现，BIRD是proxy服务器执行route advertisements的一个轻量级及可靠手段。

分割Varnish流量

Tumblelogs由它们FQDN的识别，例如一个blog的所有URI路径都会在这个blog的FQDN下发现。绝大多数的 Tumblelogs都是tumblr.com的子域，比如engineering.tumblr.com，但是Tumblr也允许用户自定义域名。

当着眼格式的 FQDN时，TLD在最小变化上有着绝对的优势，然后就是域名，子域名。因此，大部分的有效位都在域名最左端。

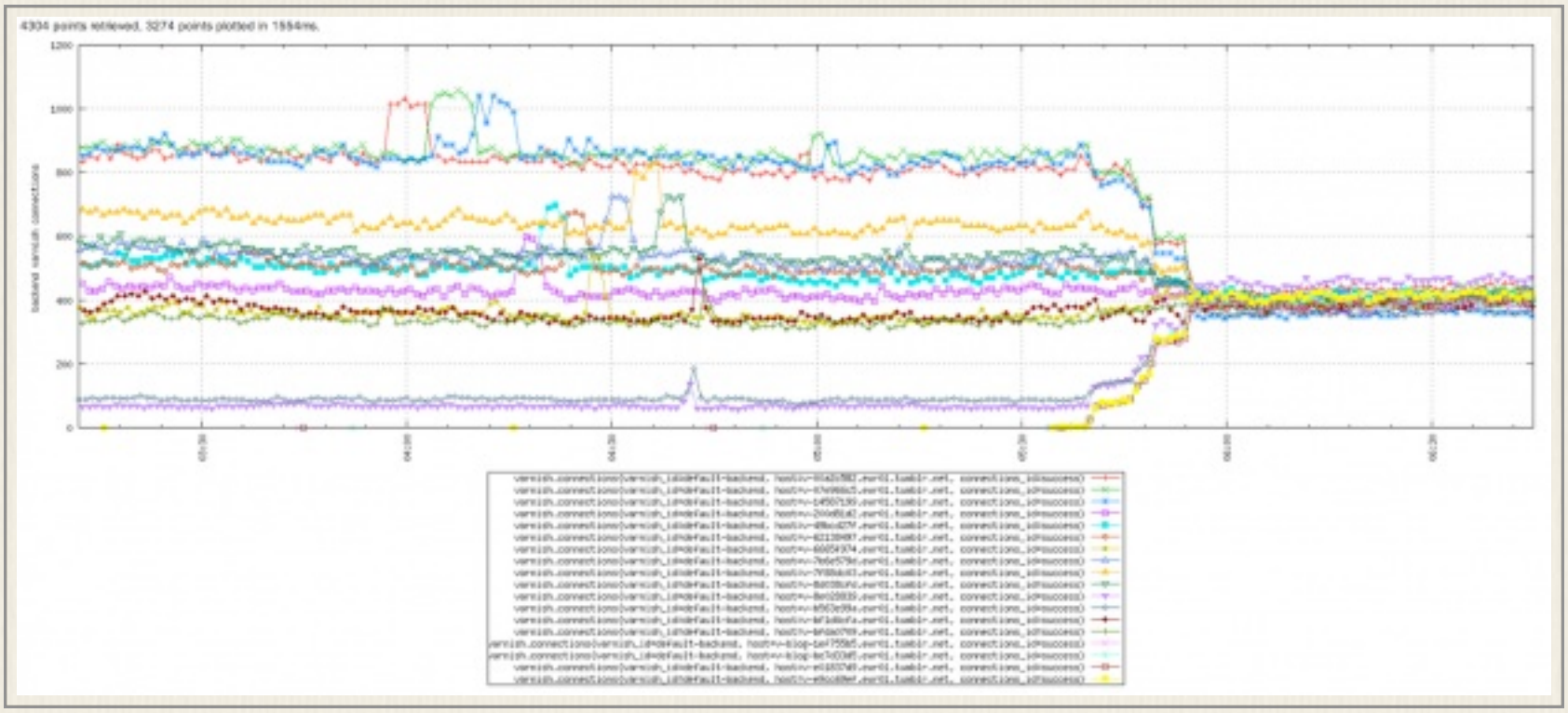
理解问题域



- 追求完美——在测试数据集上寻求最完美的哈希
- consis-tent_hdr——在主机标识上做一致性哈希（现实示例中最好结果）
- consistent_hdr_use_domain_only——在基域名上做一致性哈希（比如 tumblr.com或foo.net），只存在两种结果tumblr.com和其他
- mapbased_firstchar——将主机表示的第一个字母映射给varnish节点，这也是我们最原始的静态分割实现
- mapbased_hdr——主语主机表示映射

当一致性哈希被确立为最适合方案时，我们开始聚焦哈希函数是否合适。HAProxy默认使用的是SDBM哈希函数，然而在更深入的调查后，对比了SDBM、CRC、MD5、DJB2等，我们发现DJB2提供了更好的分布。

对比静态分割和一致性哈希



上图显示了每个varnish 节点上的变化，对比了使用最佳哈希函数前后

附加思考

节点增长

在这两种模型中，节点增长都意味着keyspace转移，因此缓存失效。在一致性哈希模型中，失效key所占的比率更加容易预测。在静态分割模型中，除下做很具体的统计，很难预测到这个百分比。

节点故障

通过静态分割，单点故障将导致 1/N的key无法访问，除非你提供一个故障转移机制。HAProxy确实允许你拥有一个备份节点，因此你需要做出 决

策，是否要为每个key space都做活跃和备份缓存节点设置，或者共享一个备份节点。一个极端意味着你将浪费50%的硬件，另一个（共享备用节点）则意味着两个故障节点就需要备用节点支撑活跃节点的2倍keyspace。

通过一致性哈希，节点故障被自动处理。当一个节点被移除后，那么1/N的key会被转移同时无效，然后把这些分配到剩余的活跃几点上。

清理缓存

清理请求可以很简单的发送到单独的varnish节点上，那么从多个varnish节点上的清理应该同样简单。取代谨慎的保持proxy和清理同 步，将所有清理请求发送到相同的proxy显然更加简单。同时，需要注意的是，拒绝不同IP空间上的清理请求非常重要，这可以防止恶意的批量清理。

学到的知识

- 有时候，问题的答案比你问题出现的更早。当面对一些扩展性挑战时，不要在那些已经在其他地方攻克过的问题上浪费时间。
- 保持简单。通过复杂性来解决扩展性问题必然深受其害。
- 理解你的哈希函数。哈希函数的选择同样至关重要。
- Degrade，not fail。让proxy具备监视自己后端到达情况的能力非常必要。如果出现异常，不要停止advertise路由，继续advertise非优先级路由。这种情况下，如果你所有的后端都出了问题，那么你仍然可以显示错误页面。

原译文链接：http://www.csdn.net/article/2014-08-05/2821037-tumblr-hashing-your-way-to-handling-23000-blog-requests-per

原文链接：http://highscalability.com/blog/2014/8/4/tumblr-hashing-your-way-to-handling-23000-blog-requests-per.html

百万用户时尚分享网站feed系统扩展实践

作者：Thierry Schellenbach

摘要：Fashiolista作为一个以社交、分享为主的在线时尚交流网站，其feed系统占据了网站的核心架构，其扩展性至关重要。网站创始人兼CTO撰写博客，分享了自家网站feed系统扩展的经验。

Fashiolista是 一个在线的时尚交流网站，用户可以在上面建立自己的档案，和他人分享自己的以及在浏览网页时看到的时尚物品。目前，Fashiolista的用户来自于全 球100多个国家，用户达百万级，每日分享的时尚物品超过500万。作为一个以社交、分享的网站，feed系统占据了网站的核心架 构，Fashiolista的创始人兼CTO Thierry Schellenbach撰写了一篇博客，分享了自家网站feed系统建设的经验，译文如下：

Fashiolista最初是我们作为兴趣在业余时间开发的一个项目，当初完全没有想到它会成长为规模如此大的在线时尚交流网站。最早的版本开发用了大概两周的时间，当时feed信息流推送系统相当简单。在这里分享一些我们扩展feed系统的经验。

对于许多大型的创业公司，如Pinterest、Instagram、Wanelo和Fashiolista来说，feed是一个核心组件。在Fashiolista网上的 flat feed、aggregated feed和notification系统功能都是靠feed系统来支撑的。本文中将介绍我们在扩展feed系统中遇到的问题，以及你自己方案中的设计决策。随着越来越多的应用依赖于feed系统，理解feed系统的基本工作原理变得至关重要了。

另外，Fashiolista的feed系统Python版本——Feedly已经开源了。

Feed简介

feed系统的扩展问题曾引起过广泛关注，这个解决方案是为了在网络拥挤的情况下，构建一个类似于Facebook新鲜事feed、Twitter流或Fashiolista的feed页面。这些系统的共同点在于向用户展示其关注的人的动态，我们就是基于这个标准来构建动态数据流的，诸如“Thierry在Fashiolista列表中添加了一件服饰”或“Tommaso发布了一条twitter”。

构建这个feed系统用到了两个策略：

1. 拉取（Pull），读取的过程中收集feed。
2. 推送（Push），写的过程中提前计算好feed。

大多数实时在线应用程序会使用这两种方法的组合，将动态推送给你的粉丝的过程被称为消息分发（fanout）。

历史和背景

Fashiolista的feed系统经过了三次重大改进。第一个版本基于PostgreSQL数据库，第二个版本使用Redis数据库，目前的版本采用Cassandra数据库。为了便于读者更好的理解这些版本更替的时间和原因，笔者会首先介绍一些背景知识。

第一部分——数据库

第一版本的数据库查询语句很简单，类似于这种：

```
select * from love where user_id in (...)
```

令 人惊讶的是这个系统的强健性还不错。当love（类似于“赞”了某件服饰）的数量达到百万时，它运行得很好，超过500万时，依然没有问题。我们还打赌说 这个系统不能支持千万的数量级，但是当love到达千万时，它依然运行得很好。这个简单的系统支撑着我们的系统达到了百万的用户和过亿的love，期间只 进行了一些小改动。之后随着用户的增多，这个系统开始出现波动，部分用户的延时长达数秒，在参考了很多关于feed系统的架构设计之后，我们开发了第一个 基于Redis的Feedly。

第二阶段——Redis和Feedly

我们为每个用户建立一个用Redis存储的feed，当你love了一件服饰时，这个动态会分发给你所有的粉丝。我们尝试了一些小技巧来减少内存的消耗（笔者会在下面具体介绍），Redis的启动和保持确实比较简单。我们使用Twemproxy在几台Redis机器上进行共享，使用Sentinel做自动备份。

Redis 是一个好的解决方案，但是几个原因迫使我们不得不寻找新的方案。首先，我们希望支持多文档类型，而Redis返回数据库查询更困难，并且提高了存储需求。 另外，随着业务的增大，数据库回滚也变得越来越慢。这些问题只能靠在Redis上存储更多的数据来解决，但是这样做的成本太高了。

第三阶段——Cassandra和Feedly

通过比较HBase、DynamoDB和Cassandra2.0，我们最终选择了Cassandra，因为它拥有几个移动部件，Instagram使用的数据库就是Cassandra，并且Datastax为 它提供支持。Fashiolista目前在flat feed中完全采取推送流，聚合feed采用推送和拉取混合的技术。我们在每个用户的feed中最多保存3600条动态，目前占用了2.12TB的存储空 间。由明星用户带来的系统波动我们也采取了一些方式进行缓解，包括：优先队列、扩大容量和自动扩展等。

Feed设计

笔者认为Fashiolista设计的改进过程非常有代表性，在构建一个feed系统时（尤其是使用Feedly）有几个重要的设计问题需要考虑。

1.非规范化Vs规范化

规范化的方法是，你关注的人的feed列表中是每条动态的ID，非规范的存储是动态的所有信息。

仅 存储ID可以大幅度减少内存消耗，然而这意味着每次加载feed都要重新访问数据库。如何选择取决于你在进行非规范化存储时，复制数据的频率。比如构建一 个消息通知系统和一个feed系统有很大的区别：通知系统

中每个动作的发生只需要被发送给几个用户，而feed系统中每个动态的数据可能要被复制给成千上 万的粉丝。

另外，如何选择取决于你的存储架构，使用Redis时，内存是需要特别注意的问题；而使用Cassandra要占用大量的存储空间，但是对于规范化数据来说使用并不简单。

对于feed通知和基于Cassandra构建的feed，笔者建议将你的数据非规范化。而基于Redis的feed你需要最小化内存消耗，并保持数据规范化。采用Feedly可以轻松实现两种方案。

2.基于生产者的选择性分发

Yahoo的Adam Silberstein等人所著的论文中，提出了一种选择性推送用户feed的方法，Twitter目 前也在使用类似的方法。明星用户的消息分发会给系统带来突然和巨大的负载压力，这意味着必须要预留出额外的空间来保持实时性。这篇论文中建议通过选择性地 分发消息，来减少这些明星用户带来的负载。Twitter采用了这个方法后，在用户读取时才加载这些明星用户的tweet，性能得到了大幅度提升。

3.基于消费者的选择性分发

另 外一种选择性分发方式是指对那些活跃用户（比如过去一周登录过的用户）分发消息。我们对这个方法进行了修改，为活跃用户存储最近的3600条动态，为非活 跃用户存储180条，读取180条之后的数据需要重新访问数据库，这种方式对于非活跃用户的体验不太好，但是能有效降低内存消耗。

Silberstein等人认为最适合选择性推送模式的情境是：

1. 生产者偶尔生产动态信息
2. 消费者经常请求feed

遗憾的是Fashiolista还不需要如此复杂的系统，很好奇业务要达到多少数量级才会需要这种解决方案。

4.优先级

一 个替代的策略是在分发任务时采取不同的优先级，将给活跃用户的分发任务设为高优先级，向非活跃用户的分发任务设为低优先级。Fashiolista

为高优 先级的用户预留了一个较大的缓存空间，来处理随时的峰值。对于低优先级用户，我们靠自动扩展和点实例。在实践中，这意味着非活跃用户的feed会有一定的 延时。使用优先级降低了明星用户对系统的负载压力，虽然没有解决根本问题，但大幅度降低了系统负载峰值的量级。

5.Redis Vs Cassandra

Fashiolista和Instagram都经历了从Redis开始，然后转战Cassandra的过程。笔者之所以会推荐从Redis开始是因为Redis更容易启动和维持。

然 而Redis存在一定的限制，所有的数据需要被存储在RAM中，成本很高。另外，Redis不支持分片，这意味着你必须在结点间分片（Twemproxy是一个不错的选择），这种分片很容易，但是添加和删除节点时的数据处理很复杂。当然你可以将Redis作为缓存，然后重新访问数据库，来克服这个限制。但 是随着访问数据库的成本越来越高，笔者建议还是用Cassandra代替Redis。

Cassandra Python的生态系统正在发生巨变，CQLEngine和Python-Driver都是很优秀的项目，但是它们需要投入一定的时间去学习。

结论

在构建自己的feed解决方案时，有很多因素需要在节点分片时考虑：选择何种存储架构？如何处理明星用户带来的负载峰值？非规范化数据到何种程度？笔者希望借助这篇文章能够为你提供一些建议。

Feedly不会为你做任何选择，这仅是一个构建feed系统的框架，你可以自己决定内部的技术细节。可以看Feedly的介绍进行了解或参看操作手册构建一个Pinterestsque应用程序。

请注意只有数据库中的用户达到百万时，你才会需要解决这个问题。在Fashiolista简单的数据库解决方案就支撑我们达到了百万用户和过亿的love。

原译文链接：http://www.csdn.net/article/2013-11-07/2817430-design-decisions-for-scaling-your-high-traffic-feeds

原文链接：http://highscalability.com/blog/2013/10/28/design-decisions-for-scaling-your-high-traffic-feeds.html